B10

BEST AVAILABLE COPY

PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 5 :
G06F 3/14, 9/00, G06K 9/36
G06K 9/46, 9/00

A1

(11) International Publication Number: WO 94/03853

(43) International Publication Date: 17 February 1994 (17.02.94)

(21) International Application Number: PCT/US93/06883

(22) International Filing Date: 22 July 1993 (22.07.93)

(30) Priority data:
07/921,831 29 July 1992 (29.07.92) US

(71) Applicant: COMMUNICATION INTELLIGENCE CORPORATION [US/US]; 275 Shoreline Drive, Redwood Shores, CA 94065 (US).

(72) Inventor: OSTREM, John, S. ; 777 San Antonio Road, #125, Palo Alto, CA 94303 (US).

(74) Agent: ALLEN, Kenneth, R.; Townsend and Townsend Khourie and Crew, One Market Plaza, 20th Fl., Steuart Tower, San Francisco, CA 94105 (US).

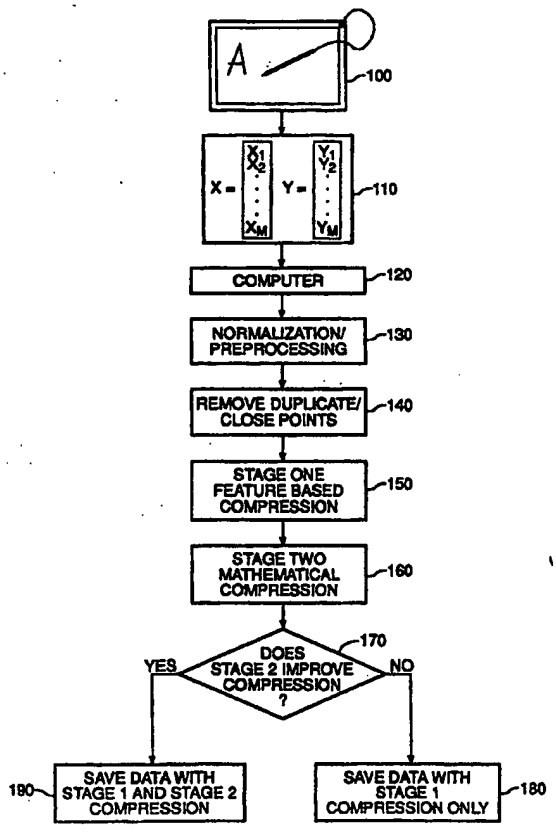
(81) Designated States: CA, DE, GB, JP, KR, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.

(54) Title: A METHOD AND APPARATUS FOR COMPRESSION OF ELECTRONIC INK

(57) Abstract

A method and apparatus are provided for compressing data received from a digitizer tablet based on the characteristics of each individual stroke contour (150, 160). The digitizer tablet samples the position of the writing pen, continuously transmitting data to a computer in the form of x and y coordinates plus an indicator of whether or not the pen is touching the surface of the tablet (100, 110). After preprocessing, a determination is made of which points along the stroke contour are essential to the reconstruction of a quality facsimile of the original writing from the sampled points (140). This determination is based upon local curvature, local extrema, and the endpoints of the stroke. In an optional second stage of compression, a standard mathematical compression technique is applied (160).



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria	**FR**	France	**MR**	Mauritania
AU	Australia	**GA**	Gabon	**MW**	Malawi
BB	Barbados	**GB**	United Kingdom	**NE**	Niger
BE	Belgium	**GN**	Guinea	**NL**	Netherlands
BF	Burkina Faso	**GR**	Greece	**NO**	Norway
BG	Bulgaria	**HU**	Hungary	**NZ**	New Zealand
BJ	Benin	**IE**	Ireland	**PL**	Poland
BR	Brazil	**IT**	Italy	**PT**	Portugal
BY	Belarus	**JP**	Japan	**RO**	Romania
CA	Canada	**KP**	Democratic People's Republic of Korea	**RU**	Russian Federation
CF	Central African Republic			**SD**	Sudan
CG	Congo	**KR**	Republic of Korea	**SE**	Sweden
CH	Switzerland	**KZ**	Kazakhstan	**SI**	Slovenia
CI	Côte d'Ivoire	**LI**	Liechtenstein	**SK**	Slovak Republic
CM	Cameroon	**LK**	Sri Lanka	**SN**	Senegal
CN	China	**LU**	Luxembourg	**TD**	Chad
CS	Czechoslovakia	**LV**	Latvia	**TG**	Togo
CZ	Czech Republic	**MC**	Monaco	**UA**	Ukraine
DE	Germany	**MG**	Madagascar	**US**	United States of America
DK	Denmark	**ML**	Mali	**UZ**	Uzbekistan
ES	Spain	**MN**	Mongolia	**VN**	Viet Nam
FI	Finland				

A METHOD AND APPARATUS FOR COMPRESSION OF ELECTRONIC INK

BACKGROUND OF THE INVENTION

The present invention relates generally to data capture applications, and more specifically, to the compression of data generated on a digitizer tablet.

When something is written on a digitizer tablet, the output is a parametric representation of the writing trajectory; that is, the writing is represented as a series of x, y coordinate values as a function of time. A typical digitizer tablet samples the position of the writing pen 100 times per second, at fixed intervals of 0.01 seconds. The tablet continuously transmits the resulting data to the receiving computer. Thus every 0.01 seconds, the tablet samples the pen position and transmits an x-coordinate value, a y-coordinate value, and an additional value which indicates whether the pen is up or down. The data generated in this manner is commonly referred to as electronic ink.

There are applications where it is of interest to capture and save the electronic ink. For example, it may be desirable to save an electronically generated signature for later signature recognition. Another application is the permanent storage of data which has been generated on a digitizer tablet.

The problem with saving electronic ink is that the storage requirements can be quite large if the original sampled data is stored directly. Therefore there is a need for a data reduction technique for electronic ink that can achieve high data compression and yet be efficient to compute. The computational efficiency is important because electronic ink may be used on computers with relatively modest CPU power. Even systems of greater power might be significantly slowed if a complex decompression technique had to be applied to each pen stroke. (Note: a "stroke" is defined as the continuous segment which begins when the pen touches the digitizer tablet's surface and ends when the pen is lifted from the surface.)

There are a number of methods for data compression. The earliest methods were based on the idea of minimum redundancy coding. The basic idea was very simple; symbols which occur frequently are coded with fewer bits than those which occur more rarely. The earliest known method to apply this technique was the Shannon-Fano coding. A subsequent development was Huffman coding, which is similar to Shannon-Fano coding in many ways. Huffman's contribution was in devising a very clever and elegant way to construct the coding and decoding trees. In general, simple Huffman coding does not achieve the same degree of compression as more modern techniques, but processing and memory requirements are relatively modest.

A further enhancement of Huffman coding is called adaptive Huffman coding. A basic problem with standard Huffman coding is that the decoding program must have access to the Huffman coding statistics. Hence, trying to increase compression with more comprehensive statistics is eventually self-defeating since more and more coding statistics must also be saved, which works against the compression. Adaptive Huffman coding gets around this problem by adjusting the Huffman decoding tree on the fly, effectively gaining the advantages of higher order modeling with no additional storage for statistics.

Huffman coding requires that an integral number of bits be used to code symbols. Arithmetic coding improves compression by using a fractional number of bits per code. Although arithmetic coding will always provide as good or better compression than Huffman coding, its disadvantage is a significantly larger computational burden.

Dictionary coding is another class of data compression. In general, these techniques create a dictionary of symbol sequences or combinations which are then accessed by indexing into the dictionary. These techniques have become the standard for compression on small computers because they combine good compression with modest memory requirements. Examples of codes using these techniques which are applicable to MS-DOS include PKZIP, ARC, ARJ, and Lharc.

There are also compression techniques that apply primarily to specialized domains. An example is compression of two dimensional images using the discrete cosine transform and its many derivatives.

A method and apparatus for compressing electronic ink which achieves high compression while requiring minimal memory and CPU power is needed.

SUMMARY OF THE INVENTION

According to the invention, for a computer with a digitizing tablet, a method and apparatus are provided for compressing data based on the characteristics of each individual stroke contour.

The digitizer tablet samples the position of the writing pen a pre-determined number of times per second and at fixed intervals. This data is continuously transmitted to a computer in the form of x and y coordinates plus an indicator of whether or not the pen is touching the surface of the tablet.

In the preferred embodiment the incoming data from the digitizer tablet is preprocessed. This preprocessing, which is application and/or system specific, is not part of the compression. Examples of preprocessing include smoothing for noise reduction, endpoint filtering, discarding duplicate data points, median filtering, Hanning filtering, spatial sampling, and size normalization.

In the first stage of compression a determination is made of which points along the stroke contour are essential to the reconstruction of a quality facsimile of the original writing from the sampled points. The method proceeds by calculating descriptors and finding essential features of the stroke contour. In the preferred embodiment, the descriptors are local curvature, local extrema, and the endpoints of the stroke. Although the actual algorithm is somewhat detailed, in general the idea is that the extrema are saved, the points about which the local curvature is above a threshold are saved, and the endpoints of the stroke are saved.

In the second stage of compression any number of standard compression techniques can be applied including Huffman coding, adaptive Huffman coding, arithmetic coding, and a variety of dictionary compression techniques. In the preferred embodiment differences of the form $\Delta x = x_i - x_{i-1}$ and $\Delta y = y_i - y_{i-1}$ are calculated and the stage 2 compression is applied to the differences.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a flow chart of the general compression method according to the invention.

Fig. 2 is a flow chart of the extrema determination step.

Fig. 3 is a flow chart of the stroke endpoint determination step.

Fig. 4 illustrates the local curvature calculations.

Fig. 5 is a flow chart of the duplicate data points preprocessing step.

Fig. 6 illustrates the endpoint filtering preprocessing step.

Fig. 7 is a flow chart of the endpoint filtering preprocessing step.

Fig. 8 is a flow chart of the median filtering preprocessing step.

Fig. 9 is a flow chart of the Hanning filtering preprocessing step.

Figs. 10A-C illustrate the reconstruction of an electronically written signature without applying this invention's compression techniques.

Figs. 11A-C illustrate the reconstruction of an electronically written signature after applying this invention's compression techniques.

Figs. 12A-C illustrate the reconstruction of a sequence of lower case letters after applying this invention's compression technique.

Figs. 13A-C illustrate the reconstruction of a simple line after applying this invention's compression technique.

Figs. 14A-E illustrate the trade off between compression ratio and reconstruction fidelity.

Fig. 15 illustrate the apparatus to accomplish the disclosed method.

DESCRIPTION OF THE SPECIFIC EMBODIMENT(S)

The flow chart of Fig. 1 illustrates the compression method described by this invention. Data is input by writing on a digitizer tablet (step 100). The tablet represents the writing as a series of x and y coordinate values as a function of time (step 110). This stream of x,y position coordinates along with a pen up/pen down indicator for each coordinate pair is continuously sent to the computer (step 120).

In the preferred embodiment, prior to data compression, the data is preprocessed (step 130). The exact form of preprocessing will vary according to the application and system. Preprocessing is generally used to remove noise. The details are unimportant as long as a good representation of the written input data is maintained. Another optional step which is dependent upon the application and system is the removal of duplicate or close points (step 140). This step is generally necessary when the digitizer tablet's user writes slowly. Slow writing causes oversampling resulting in duplicate or close points which are redundant and do not carry significant information.

The first stage of compression is based on the characteristics of each individual stroke contour (step 150). In the second stage of compression any of the standard mathematical compression techniques can be applied (step 160). Although this stage of compression is not required by the invention, it is employed in the preferred embodiment as a potential means of further compressing the data.

After the completion of the second stage, a comparison is made between the first stage and the second stage compression (step 170). If the compression was not improved after applying stage two, then the stage one compression data is saved (step 180). If the data was further compressed by stage two, then this compression data is saved (step 190).

The computer program modules used to compress the electronic data are given in the appendix. Significant program modules and their specific functions are as follows:

Stage one compression:

calc_sav.c	Function to determine which points in the signature need to be saved, based on various criteria.
curvature.c	Function to calculate the local curvature at a point along the contour.

Stage two compression:

cmpsig2.c	Function to compress an array of stage 1 data with characteristics of the signature verification template data.
pack_halfbytes.c	Function to pack the value to be compressed into half bytes.
set_4_bits	Function to pack values into the upper or lower 4 bits of a byte.
get_4_bits	Function to unpack values from the upper or lower 4 bits of a byte.
uncmpsig2.c	Function to uncompress an array of data compressed by the function cmpsig2.c.
unpack_halfbytes.c	Function to unpack and reconstruct the value from half bytes.

Figs. 2-4 are detailed illustrations of the stage one compression methods. Fig. 2 shows a flow chart of the first step of compression in which extrema are determined and saved. The digitizer data is in the form of a sequence of coordinate values, x_i and y_i, as a function of time (step 210). The subscript i varies from i=0 to i=nsamps-2 where nsamps is the total number of data points in a stroke. The subroutine is initialized by setting i equal to 1 (step 220).

Minima and maxima are determined by comparing the value of each data point to the values of the data points directly adjacent to it (step 230). To be a maxima, the value

of either the x coordinate or the y coordinate must be larger then the values of the corresponding coordinate of the data points on both sides of the data point under review (step 230). To be a minima, the value of either the x coordinate or the y coordinate must be smaller then the values of the corresponding coordinate of the data points on both sides of the data point under review (step 230).

If any of the four relationships defined by step 230 are met (step 240), then that data point will be saved (step 250), otherwise the data point is not saved. After determining whether or not to save a data point, the value of i is increased by one (step 260) and a determination is made as to whether or not all data points have been reviewed (step 270). If all data points have been reviewed then this subroutine ends (step 280), otherwise the subroutine loops back to the comparison step (step 230), continuing until all data points have been reviewed.

Fig. 3 shows a flow chart of the second step of compression in which stroke endpoints are determined and saved. The digitizer data is in the form of a sequence of coordinate values, x_i and y_i, as a function of time (step 310). The subscript i varies from i=0 to i=nsamps-1 where nsamps is the total number of data points in a stroke. The subroutine is initialized by setting i equal to 0 (step 315).

This subroutine first determines whether or not the x and y coordinates of the data point indicate that it is the first data point of the stroke (step 320). If it is the first data point, then it is saved (step 330). Next the subroutine determines whether or not the x and y coordinates of the data point indicate that it is the last data point of the stroke (step 340). If it is the last data point of the stroke, then it is saved (step 350). After determining whether or not to save the data point, a comparison is made to determine if all data points have been tested (step 360). If they have been then this subroutine ends (step 370), otherwise the value of i is increased by one (step 380) and endpoint determination continues.

Fig. 4 illustrates the calculations used to determine the curvature at each data point between the stroke endpoints. The stroke contour is represented by line 410 while the individual digitizer data points are represented by points 420. Point 430 is data point i, the data point of interest in this illustration. The coordinates of 430 are x_i and y_i.

In Fig. 4 line 440 is defined as the line segment connecting point 430 with point 425. The coordinates of point 425 are x_{i-d} and y_{i-d}, where d is equivalent to 3. Line 450 is defined as the line segment connecting point 430 to point 435 where point 435 has coordinates x_{i+d} and y_{i+d}. Curvature angle 460 is the angle defined by lines 440 and 450.

During stage one compression, the curvature angle for each data point is calculated. If the calculated curvature angle is within a certain range of angles, the data point is saved. If the angle does not fall within the pre-selected range of angles, the data point is discarded. The choice of the variable "d" as well as the range of threshold angles is determined by the application's requirements. In the preferred embodiment, the value of d is 2; data points which have a curvature angle between 5 degrees and 25 degrees are retained for applications requiring high fidelity, low compression (e.g., signature verification); and data points which have a curvature angle between 25 and 50 degrees are retained for applications requiring low fidelity, high compression (e.g., simple data storage).

Figs. 5-8 are detailed illustrations of optional data preprocessing methods. Fig. 5 is a flow chart of the method by which duplicate data points are discarded. The digitizer data is in the form of a sequence of coordinate values, x_i and y_i, as a function of time (step 510). The subscript i varies from i=0 to i=nsamps-1 where nsamps is the total number of data points in a stroke. The subroutine is initialized by setting i equal to 1, x_{sav} equal to x_0 and y_{sav} equal to y_0 (step 520).

The current data point, represented by x_i and y_i, is compared to the previous data point, represented by x_{sav} and y_{sav} (step 530). If the values for the x and y coordinates are identical, then the current point is discarded (step 540) and

the next point is compared to x_{sav} and y_{sav}. This process continues until either all of the data points have been reviewed (step 560) or until x_i and y_i do not equal x_{sav} and y_{sav}, respectively. When the two data points are no longer equivalent, x_{sav} is set equal to x_i and y_{sav} is set equal to y_i (step 545). The value of i is then increased by one (step 550) and the new value of i is compared to the total number of data points (step 560). The subroutine continues until all data points have been reviewed at which time the subroutine ends (step 570).

Often the beginning and ending of a stroke have spurious artifacts which can be removed with little effect on the writing's fidelity. These artifacts can be due to the writer's unsureness, resulting in wiggling of the pen upon contact, or due to pen skidding. Fig. 6 illustrates the preprocessing method of endpoint filtering which can be used to remove these artifacts.

Stroke contour 610 is made up of a series of data points 620. Circle 630, centered at the stroke's first data point 635, and circle 640, centered at the stroke's last data point 645, are used to define the threshold by which points are either discarded or saved. In the preferred embodiment, the physical radius of this circle is set at 0.5 millimeters. The number of data points contained within these circles is dependent upon the resolution of the digitizer tablet. The resolution of a typical tablet is 10 points per millimeter. Therefore the threshold, t, given in tablet units is obtained by multiplying the desired physical distance (0.5 millimeters) by the tablet's resolution (10 points/millimeter).

Fig. 7 is a flow chart for the endpoint filtering preprocessing method. The digitizer data is in the form of a sequence of coordinate values, x_i and y_i, as a function of time (step 710). The subscript i varies from i=0 to i=nsamps-1 where nsamps is the total number of data points in a stroke. The subroutine is initialized by setting i equal to 0 (step 715) and setting j equal to 1 (step 720).

The variable d is calculated (step 730) and compared to the pre-selected threshold, t (step 740). If d is less than t,

indicating that this point is within circle 630, the point corresponding to x_j, y_j is discarded (step 750). The value of j is then increased by one (step 760) and the calculation for d (step 730) is repeated. This process continues until d is not less then the threshold value, at which time that value of x_j, y_j is saved as the second data point (step 770). This completes the subroutine for endpoint filtering for the beginning of a stroke.

The endpoint filtering method used for the last point of a stroke is identical to that used for the first point except that to initialize the subroutine i is set equal to nsamps-1 (step 715) and j is set equal to nsamps-2 (step 720).

Median filtering is a preprocessing method used in the preferred embodiment to reduce the effects of tablet glitches that result in sudden jumps or discontinuities in the data stream. Fig. 8 is a flow chart illustrating this method. The digitizer data is in the form of a sequence of coordinate values, x_i and y_i, as a function of time (step 810). The subscript i varies from i=1 to i=nsamps-2 where nsamps is the total number of data points in a stroke. The subroutine is initialized by setting i equal to 1 (step 820).

In this preprocessing method, a new value for each coordinate is calculated based on the median value of three successive data points (steps 830 and 840). The value of i is then increased by one (step 850) and the subroutine continues until the value of i is equal to the total number of samples minus two (step 860) at which time the subroutine is complete (step 870).

A flow chart of the last preprocessing method used in the preferred embodiment is illustrated in Fig. 9. The digitizer data is in the form of a sequence of coordinate values, x_i and y_i, as a function of time (step 910). The subscript i varies from i=1 to i=nsamps-2 where nsamps is the total number of data points in a stroke. The subroutine is initialized by setting i equal to 1 (step 920).

In this method, a linear process is applied to reduce noise and smooth the stroke contour. A new value for each x and y coordinate is calculated based on the initial values of

three successive data points (step 930). The value of i is then increased by one (step 940) and the subroutine continues until the value of i is equal to the total number of samples minus two (step 950) at which time the subroutine is complete (step 960).

Fig. 10A is an original signature. Fig. 10B shows the actual points received from the digitizer tablet. Fig. 10C is a reconstruction of the signature based upon the points shown in Fig. 10B.

Fig. 11A is another original signature. Fig. 11B, as in Fig. 10B, shows the actual points received from the digitizer tablet and are shown as single points 1110. At this point, prior to compression, the signature requires 1088 bytes of storage. The points 1120 which have been circled are those points which have been determined to be critical to the reconstruction fidelity. Fig. 11C is the reconstructed signature based upon the circled points 1120. This signature required only 219 bytes of storage, or 20 percent of the original. The stage one compression reduced the 1088 bytes to 584 bytes, which was further reduced to 219 bytes by applying a stage two compression technique.

Fig. 12A is the original of a sequence of lower case letters. As in Fig. 11B, Fig. 12B shows both actual points 1210 received from the digitizing tablet and points 1220 determined to be critical to the reconstruction fidelity. Fig. 12C is the reconstruction based upon circled points 1220. The original required 2072 bytes, the reconstruction based upon stage one compression only required 800 bytes (39 percent of the original), and the reconstruction shown in 12C in which both stage one and stage two compression was applied required 301 bytes (15 percent).

Fig. 13A-C illustrates that the algorithm automatically adjusts to the complexity of the electronic ink. Fig. 13A is the original of a straight line which contains little structure. Fig. 13B shows both actual points 1310 received from the digitizing tablet and points 1320 determined to be critical to the reconstruction fidelity. Fig. 13C is the reconstruction based upon circled points 1320 in which only

stage one compression has been applied. The reconstruction required only 6 bytes and represented a compression to 2 percent of the original. In this case stage two compression was not applied because it increased the number of bytes to 14.

Figs. 14A-E illustrates the trade between compression ratio and the fidelity of the reconstruction. Fig. 14A is the original of a printed address which required a total of 1808 bytes. Fig. 14B shows the first level of compression which employed a compression ratio similar to that used in Figs. 11C and 12C. After both stage one and stage two compression, this reconstruction is 13 percent of the original. Figs. 14C-E show gradually increasing levels of compression. The reconstruction shown in Fig. 14C, requiring only 9 percent of the original, appears quite similar to the original. Figs. 14D-E, although quite readable, show definite differences from the original. The reconstructions in Figs. 14D and 14E require 8 percent and 7 percent of the original storage space, respectively. Although some applications require high fidelity, such as signature verification, many applications could tolerate the lower fidelity accompanying the high compression ratios shown in Figs. 14D-E.

Fig. 15 shows an embodiment of the apparatus to accomplish the method described in this invention. In the preferred embodiment, the computer 1550 is an IBM/PC/AT compatible with an IBM compatible monitor 1540 and keyboard 1510. Computer 1550 employs the methods disclosed by this invention to compress the electronic ink. Handwritten information is input either directly via the hardwired digitizer tablet 1520 or via a portable digitizer tablet 1530. Portable digitizer tablets are in common use in many industries, including the package delivery industry where an electronic signature is obtained upon delivery of a package. Depending upon the computing power of tablet 1530, the signature can either be immediately compressed using this invention's compression methods or temporarily stored for subsequent downloading to computer 1550. Once the handwritten data has been compressed, it can either be stored in computer 1550 or in separate memory 1560. Monitor 1540 allows the

original handwritten data to be compared with the compressed data.

The invention has now been explained with reference to specific embodiments. Other embodiments will be apparent to those of ordinary skill in this art in light of this disclosure. It is therefore not intended that this invention be limited except as indicated by the appended claims.

APPENDIX

The computer program modules used to compress the electronic data are given in this appendix. Significant program modules and their specific functions are as follows:

Stage one compression:

calc_sav.c	Function to determine which points in the signature need to be saved, based on various criteria.
curvature.c	Function to calculate the local curvature at a point along the contour.

Stage two compression:

cmpsig2.c	Function to compress an array of stage 1 data with characteristics of the signature verification template data.
pack_halfbytes.c	Function to pack the value to be compressed into half bytes.
set_4_bits	Function to pack values into the upper or lower 4 bits of a byte.
get_4_bits	Function to unpack values from the upper or lower 4 bits of a byte.
uncmpsig2.c	Function to uncompress an array of data compressed by the function cmpsig2.c.
unpack_halfbytes.c	Function to unpack and reconstruct the value from half bytes.

```
/* CALC_SAV.C
 *-------------------------------------------------------------
 * Function to determine which points in the signature need to
```

```
 * be saved, based on various criteria.  The results are
 * returned in array save_ptsC.
 * For example, if the ith point of x[],y[] should be saved
 * then save_ptsC[i] = 1, else save_pts[i] = 0;
 * cur_comp returns:  1  (i.e. YES or  TRUE) for successful
 *                       scaling
 *                    0  (i.e. NO  or FALSE) if there is an
 *                       error
 *
 * input arguments:   curvature_threshold, nsamps, numsegs,
 * segpnt, x, y
 *
 * output arguments:  save_pts
 *-----------------------------------------------------------
 */
#include <stdio.h>
#include <cic.h>
BOOL calc_sav(curvature_threshold, nsamps, numsegs, segpnt, x,
        y, save_ptsC)
    COUNT curvature_threshold;     /* threshold for curvature*/
                                   /* compression            */
    COUNT nsamps;                  /* number of samples      */
                                   /* (length) in input      */
                                   /* arrays x and y         */
    COUNT numsegs;                 /* number of individual   */
                                   /* segments (strokes)     */
    COUNT segpnt[];                /* pointers to beg of     */
                                   /* individual strokes     */
    COUNT x[];                     /* input array of x       */
                                   /* values                 */
    COUNT y[];                     /* input array of y       */
                                   /* values                 */
    TINY  save_ptsC[];             /* contains information   */
                                   /* about what points to   */
                                   /* keep (returned to      */
                                   /* calling program)       */
    COUNT i;                       /* loop counter           */
    COUNT k;                       /* loop counter           */
```

```
        COUNT value1;                   /* temporary variable     */
        COUNT value2;                   /* temporary variable     */
        COUNT checkB;                   /* temporary variable     */
        COUNT lower_limit;              /* defines starting pt    */
                                        /* for curvat compress    */
        COUNT upper_limit;              /* defines ending pt for  */
                                        /* curvat compress        */
        BOOL  curvature();              /* function to calculate  */
                                        /* local curvature        */
        COUNT size_angle;               /* calculated curvature   */
                                        /* at a point             */
        BOOL  clockwiseB;               /* YES if clockwise, NO   */
                                        /* otherwise              */
        BOOL  calc_dir;                 /* if YES calculate       */
                                        /* dir_angleP             */
        COUNT dir_angle;                /* direction of curvature */
                                        /* bisector               */
    *-------------------------------------------------------------
    * Initialize the save_ptsC array.
     *-------------------------------------------------------------
     */
        for (i = 0; i < nsamps; ++i)
            {
            save_ptsC[i] = 1;
            }
    /*
     *-------------------------------------------------------------
     * Perform curvature based compression (determines which points
     * to keep of the stroke and saves the information in array
     * save_ptsC to return to the calling program).
     *-------------------------------------------------------------
     */
        for (i = 0; i < numsegs; ++i)
            {
            lower_limit = segpnt[i];
            upper_limit = segpnt[i + 1];
            while (lower_limit < upper_limit)
                {
```

```
                checkB = NO;
                for (k = lower_limit + 2; k < upper_limit; k += 2)
                    {
                    curvature(x, y, (k + lower_limit) / 2,
                              nsamps, (k - lower_limit) / 2,
                              &size_angle, &clockwiseB, &calc_dir,
                              &dir_angle);
                    if (size_angle > curvature_threshold)
                        {
                        checkB = YES;
                        break;
                        }
                    }
                if (checkB)
                    {
                    value2 = (k + lower_limit) / 2;
                    value1 = value2;
                    }
                else
                    {
                    value1 = segpnt[i + 1] - 1;
                    value2 = (value1 + lower_limit) / 2;
                    }
                for (k = lower_limit + 1; k < value1; ++k)
                    {
                    if (k != value2)
                        save_ptsC[k] = 0;
                    }
                if (checkB)
                    lower_limit = value2;
                else
                    lower_limit = upper_limit;
                }
            }

    /*
     *-------------------------------------------------------------
     * Return to calling program with successful completion flag.
```

```
    *-------------------------------------------------------------
    */
       return(YES);
    /*
    *-------------------------------------------------------------
    */
       }                                /* end of function        */
                                        /* CALC_SAV               */
    /* CURVATURE.C
    *-------------------------------------------------------------
    * Calculates the curvature, direction of curvature (angle)
    * bisector, and whether the change in angle is clockwise or
    * counter_clockwise at a point defined by function argument
    * "index" into the arrays of coordinates points x[], y[],
    * representing the stroke contour.  The quantities returned
    * are in terms of the angle range 0-360 degrees.
    *
    * If ERROR is detected the BOOL function value is YES (=1),
    * while for a normal return the BOOL function value is NO
    * (=0).
    *
    * CAUTION: when the curvature is 0 (i.e., calculated
    * size_angleP = 0 or close to 0), the calculation of the
    * direction of the curvature bisector is, of course,
    * inherently ambiguous and unreliable.
    *-------------------------------------------------------------
    */
    #include <stdio.h>
    #include <cic.h>

    BOOL curvature(x, y, index, maxpts, lghseg, size_angleP,
              clockwiseBP, calc_dirB, dir_angleP)
       COUNT x[];                    /* x coordinate values of */
                                     /* character              */
       COUNT y[];                    /* y coordinate values of */
                                     /* character              */
       COUNT index;                  /* point in x, y to calc. */
                                     /* curvature              */
```

```
        COUNT maxpts;                   /* maximum length of      */
                                        /* arrays x,y             */
        COUNT lghseg;                   /* length of segment for  */
                                        /* curvatur calc          */
        COUNT *size_angleP;             /* calculated curvature   */
                                        /* at point index         */
        BOOL  *clockwiseBP;             /* YES if clockwise, NO   */
                                        /* otherwise              */
        BOOL  calc_dirB;                /* if YES calculate       */
                                        /* dir_angleP, otherwise  */
                                        /* otherwise return       */
                                        /* dir_angle = 0          */
        COUNT *dir_angleP;              /* direction of curvature */
                                        /* bisector               */
        {
        COUNT plus_point;               /* stores "index + lghseg"*/
        COUNT minus_point;              /* stores "index - lghseg"*/
        COUNT plus_angle;               /* stores calc direction  */
                                        /* of upper seg.          */
        COUNT minus_angle;              /* stores calc direction  */
                                        /* of lower seg.          */
        COUNT dif_angle;                /* stores angle angle     */
                                        /* difference             */
        COUNT cic_atan2();              /* function to calculate  */
                                        /* angle in the           */
                                        /* range -1800 to 1800    */
                                        /* (degrees x 10)         */
    /*
     *-------------------------------------------------------------
     * Check for errors in specifying the range for curvature
     * computation.
     *-------------------------------------------------------------
     */
        minus_point = index - lghseg;
        plus_point  = index + lghseg;
        if (minus_point < 0 || plus_point >= maxpts)
            {
            *size_angleP = 0;
```

```
        *clockwiseBP = YES;
        *dir_angleP  = 0;
        return (YES);                   /* ERROR - out of bounds, */
                                        /* cannot calc. curvature */
        }
    /*
     *-------------------------------------------------------------
     * Calculation of the local curvature (i.e., angle change).
     * Note: the function cic_atan2 returns angle values in the
     * range -1800 to 1800, so it is necessary to divide by 10 to
     * get the desired curvatures.
     *-------------------------------------------------------------
     */
        plus_angle  = cic_atan2(x[plus_point] - x[index],
                                y[plus_point] - y[index]);
        minus_angle = cic_atan2(x[index] - x[minus_point],
                                y[index] - y[minus_point]);
        dif_angle   = plus_angle - minus_angle;
        if (dif_angle > 1800)
            dif_angle -= 3600;
        else if (dif_angle < -1800)
            dif_angle += 3600;
        if (dif_angle < 0)
            {
            *size_angleP = (-dif_angle + 5) / 10;
            *clockwiseBP = YES;         /* by convention          */
                                        /* clockwise              */
            }                           /* angle changes are      */
                                        /* negative               */
        else
            {
            *size_angleP = (dif_angle + 5) / 10;
            *clockwiseBP = NO;          /* by convention          */
                                        /* counter-clockwise      */
            }                           /* angle changes are      */
                                        /* positive               */
    /*
     *-------------------------------------------------------------
```

```
    * Calculate the direction of the bisector of the local
    * curvature (i.e., angle change) computed above in the range
    * 0-360 degrees.  Note: the cic_atan2 returns angle values in
    * the range -1800 to 1800, so an adjustment is made in the
    * code below so that the resulting calculated bisector value
    * is in the range 0-360.
    *-------------------------------------------------------------
    */
       if (calc_dirB)
           {
           if (*clockwiseBP == YES)
               *dir_angleP  = plus_angle - (900 + (dif_angle - 1)
                            / 2);
           else
               *dir_angleP  = plus_angle + (900 - (dif_angle + 1)
                            / 2);
           if (*dir_angleP < 0)
               *dir_angleP += 3600;
           *dir_angleP = (*dir_angleP + 5) / 10;
           if (*dir_angleP >= 360)
               *dir_angleP -= 360;
           }
       return(NO);                          /* normal return - no     */
                                            /* errors detected        */
    *-------------------------------------------------------------
    */
       }                                    /* end of function        */
                                            /* CURVATURE.C            */
    /*
    CMPSIG2.C
     *-------------------------------------------------------------
     * Function to compress an array of data with charcteristics of
     * the signature verification template data.  The companion
     * program to uncompress the data is UNCMPSIG2.C.  NOTE:
     * compression/uncompression is exact (not lossy) and
     * restores the original data exactly.
     *
     * cmpsig2 returns:  1  (i.e. YES or  TRUE) for successful
```

```
     *                          compression
     *                      0   (i.e. NO  or FALSE) if there is an
     *                          error
     *
     * input arguments:
     *        numin  - number of elements (length) of data array to
     *                 be compressed
     *        array  - data array to be compressed
     *        maxout - maximum allowed size of the compressed array
     *                 comp_arrayUC[] (to prevent overwriting
     *                 memory)
     *
     * output arguments:
     *        num_comp_bytesP - ptr to # elements (each a byte) of
     *                 the compressed data stored in
     *                 comp_arrayUC[].
     *        comp_arrayUC - unsigned byte array containing the
     *                 compressed representation of the input
     *                 data in array[];
     *-----------------------------------------------------------
     */
    #include <stdio.h>
    #include <cic.h>

    BOOL cmpsig2(numin, array, maxout, num_comp_bytesP,
                 comp_arrayUC)
        COUNT numin;              /* number of points in input  */
                                  /* array                      */
        COUNT array[];             /* array containing input    */
                                  /* data to be compressed      */
        COUNT maxout;             /* max number of pts for      */
                                  /* comp_arrayUC               */
        COUNT *num_comp_bytesP;   /* actual #bytes used in      */
                                  /* comp_arrayUC               */
        TBITS comp_arrayUC[];     /* output array of compressed */
                                  /* data                       */
        {
        COUNT i;                  /* loop index                 */
```

```
        COUNT num_sign_change;    /* counts number of sign      */
                                  /* changes                    */
        COUNT value;               /* for general use temporary */
                                  /* storage                    */
        COUNT byte_part;          /* 1 if upper 4 bits of byte, */
                                  /* and 2 if lower 4 bits of   */
                                  /* byte                       */
        TBITS *comp_arrayUCP;     /* ptr to comp_arrayUC of sig.*/
                                  /* data                       */
        TBITS byte_maskUC;        /* byte used to temporarily   */
                                  /* hold compressed data       */
        COUNT last_index;         /* saves last index of direct.*/
                                  /* change                     */
        COUNT current_sign;       /* used in sign change        */
                                  /* determination              */
        COUNT last_sign;          /* used in sign change        */
                                  /* determination              */
        BOOL  set_4_bits();       /* function to pack values    */
                                  /* into the upper or lower    */
                                  /* 4 bits of a byte           */
        BOOL  get_4_bits();       /* function to unpack values  */
                                  /* from the upper or lower    */
                                  /* 4 bits of a byte           */
        BOOL  pack_halfbytes();   /* implements main packing    */
                                  /* logic                      */
    /*
     *-------------------------------------------------------------
     * Initializations for the compression process.
     *-------------------------------------------------------------
     */
        *num_comp_bytesP = 0;
        *comp_arrayUCP    = comp_arrayUC;
    /*
     *-------------------------------------------------------------
     * First, save the number of items in the array to be
     * compressed.  This value is stored as two bytes.  The first
     * byte is the upper 8 bits of the 16 bit COUNT value for
     * numin, and the second byte is the lower 8 bits.
```

```
    *-------------------------------------------------------------
    */
        *num_comp_bytesP += 1;
        *comp_arrayUCP++  = (TBITS) ((numin >> 8) & 0xff);
        *num_comp_bytesP += 1;
        *comp_arrayUCP++  = (TBITS) (numin & 0xff);
    /*
    *-------------------------------------------------------------
    * Find the number of sign changes (i.e., when successive
    * difference values have different signs) and save to
    * comp_arrayUC.  Note that the maximum number of sign changes
    * allowed is 255.  The function immediately exits and returns
    * an error flag if > 255 sign changes are detected.
    *-------------------------------------------------------------
    */
        num_sign_change = 0;
        last_sign       = array[1] - array[0];
        for (i = 2; i < numin; ++i)
            {
            current_sign = array[i] - array[i - 1];
            if ((current_sign <  0 && last_sign >= 0) ||
                (current_sign >= 0 && last_sign <  0)   )
                {
                last_sign = current_sign;
                ++num_sign_change;
                }
            }
        if (num_sign_change < 256)
            {
            *num_comp_bytesP += 1;
            *comp_arrayUCP++  = (TBITS) num_sign_change;
            }
        else
            {
            fprintf(stderr, "\n\nCMPSIG -- num_sign_change >
                    256\n\n");
            return(NO);
            }
```

```
    /*
     *-------------------------------------------------------------
     * Initializations for compression
     *-------------------------------------------------------------
     */
        byte_part   = 1;
        byte_maskUC = 0;
    /*
     *-------------------------------------------------------------
     * Pack the sign of the first value of the input data (i.e., of
     * array[0])
     *-------------------------------------------------------------
     */
        if (array[0] >= 0)
                value = 1;
        else
                value = 0;
        pack_halfbytes(value, &byte_part, &byte_maskUC,
                    &comp_arrayUCP, num_comp_bytesP);
    /*
     *-------------------------------------------------------------
     * Pack the sign of the first difference value (i.e., of
     * array[1] - array[0])
     *-------------------------------------------------------------
     */
        last_sign = array[1] - array[0];
        if (last_sign < 0)
            value = 0;
        else
            value = 1;
        pack_halfbytes(value, &byte_part, &byte_maskUC,
                    &comp_arrayUCP, num_comp_bytesP);
    /*
     *-------------------------------------------------------------
     * Pack the sign change positions (where successive difference
     * values have different sign).
     *-------------------------------------------------------------
     */
```

```
    last_index = 0;
    for (i = 2; i < numin; ++i)
        {
        current_sign = array[i] - array[i - 1];
        if ((current_sign <  0 && last_sign >= 0) ||
            (current_sign >= 0 && last_sign <  0)   )
            {
            last_sign = current_sign;
            if ((i - last_index) > 255)
                {
                fprintf(stderr, "\n\nCMPSIG -- sign index
                        change > 255\n\n");
                return(NO);
                }
            pack_halfbytes(i - last_index, &byte_part,
                        &byte_maskUC, &comp_arrayUCP,
                        num_comp_bytesP);
            last_index = i;
            }
        }                       /* end of loop to pack sign   */
                                /* change position values     */
/*
 *-------------------------------------------------------------
 * Pack differences values into bytes and save to comp_arrayUC
 *-------------------------------------------------------------
 */
    for (i = 0; i < numin; ++i)
        {
        if (i == 0)
            value = abs(array[0]);
        else
            value = abs(array[i] - array[i - 1]);
        pack_halfbytes(value, &byte_part, &byte_maskUC,
                    &comp_arrayUCP, num_comp_bytesP);
        if (*num_comp_bytesP >= maxout)
            {
            fprintf(stderr,"\n\nCMPSIG -- comp_arrayUC too
                    large\n\n");
```

```
            return(NO);
            }
        }                       /* end of loop to pack        */
                                /* difference values          */
/*
 *-------------------------------------------------------------
 * Finally, if the data ends on an odd number of half bytes,
 * must save one final byte because the last signature data
 * value is in the first half byte of byte_maskUC and hasn't
 * yet been saved to comp_arrayUC.
 *-------------------------------------------------------------
 */
   if (byte_part == 2)
       {
       set_4_bits((UTINY) 15, 2, &byte_maskUC);
       *num_comp_bytesP += 1;
       if (*num_comp_bytesP > maxout)
           {
           fprintf(stderr, "\n\nCMPSIG -- comp_arrayUC too
              large\n\n");
           return(NO);
           }
       *comp_arrayUCP++ = byte_maskUC;
       }
/*
 *-------------------------------------------------------------
 * If this point is reached the compression process has
 * proceeded normally and no errors have been detected, so
 * return with a "successful completion" status flag.
 *-------------------------------------------------------------
 */
   return(YES);
 *-------------------------------------------------------------
 */
   }                            /* end of function CMPSIG2    */
/* PACK_HALFBYTES.C
 *-------------------------------------------------------------
 * Function to pack a COUNT value into half bytes.
```

```
    *
    *      pack_halfbytes returns:  1  (i.e. YES or  TRUE) for
    *                                  successful packing
    *                               0  (i.e. NO  or FALSE) if
    *                                  there is an error
    *-------------------------------------------------------------
    */
   BOOL pack_halfbytes(value, byte_partP, byte_maskUCP,
            comp_arrayUCPP, num_comp_bytesP)
      COUNT value;              /* value to be compressed     */
      COUNT *byte_partP;        /* 1 if upper 4 bits of byte, */
                                /* and 2 if lower 4 bits of   */
                                /* byte                       */
      TBITS *byte_maskUCP;      /* packed byte representing   */
                                /* different values           */
      TBITS **comp_arrayUCPP;   /* ptr to ptr to comp_arrayUC */
                                /* element                    */
      COUNT *num_comp_bytesP;   /* actual #bytes used in      */
                                /* comp_arrayUC               */
      {
      UTINY tempUC;             /* temporary value            */
      UTINY continuation_labelUC;/* used to save continuation*/
                                /* label                      */
      BOOL  get_4_bits();       /* function to unpack values  */
                                /* from the upper or lower 4  */
                                /* bits of a byte             */
      BOOL  set_4_bits();       /* function to pack values    */
                                /* into the upper or lower 4  */
                                /* bits of a byte             */
      if (value < 14)
          {
   /*
    *-------------------------------------------------------------
    * Pack 4 bit value into byte.  When the absolute difference
    * value is < 14, then it can be packed into a half_byte (4
    * bits).
    *-------------------------------------------------------------
    */
```

```
            set_4_bits((UTINY) value, *byte_partP, byte_maskUCP);
            if (*byte_partP == 2)
                {
                *byte_partP       = 1;
                *num_comp_bytesP += 1;
                **comp_arrayUCPP  = *byte_maskUCP;
                ++(*comp_arrayUCPP);
                }
            else
                *byte_partP = 2;
            }
        else
            {
           /*
    *-------------------------------------------------------------
    * Pack 4 bit continuation label into byte.  When the absolute
    * difference value is <= 30 (and, of course, >= 14 since the
    * previous block above deals with values < 14), then pack a
    * value 14 into the next half byte, and write the remainder
    * into a subsequent half byte. If the absolute difference
    * value is  >= 30, pack 15 into the first half byte and pack
    * the remainder into another byte.  If the absolute difference
    * value is > 14 + 255 = 269, then an error has occurred and
    * the function immediately exits and returns an error
    * condition.
    *-------------------------------------------------------------
    */
            if (value > 269)
                {
                fprintf(stderr, "\n\nCMPSIG - abs dif value out of
    ");
                fprintf(stderr, "range (= %d)\n\n", value);
                return(NO);          /* return on ERROR detection */
                }
            else if (value < 30)
                continuation_labelUC = (UTINY) 14;
            else
                continuation_labelUC = (UTINY) 15;
```

```
    /*
     *-------------------------------------------------------------
     * First, set half byte for continuation label.
     *-------------------------------------------------------------
     */
            set_4_bits(continuation_labelUC, *byte_partP,
                    byte_maskUCP);
            if (*byte_partP == 2)
                {
                *byte_partP       = 1;
                *num_comp_bytesP += 1;
                **comp_arrayUCPP  = *byte_maskUCP;
                ++(*comp_arrayUCPP);
                }
            else
                *byte_partP = 2;
            if (value < 30)
                {
    /*
     *-------------------------------------------------------------
     * Pack a value corresponding to (value - 14) into a second
     * half byte (i.e., a value from 14 to 29 will be converted to
     * a value from 0 to 15 for packing).
     *-------------------------------------------------------------
     */
                set_4_bits((UTINY) (value - 14), *byte_partP,
                        byte_maskUCP);
                if (*byte_partP == 2)
                    {
                    *byte_partP       = 1;
                    *num_comp_bytesP += 1;
                    **comp_arrayUCPP  = *byte_maskUCP;
                    ++(*comp_arrayUCPP);
                    }
                else
                    *byte_partP = 2;
                }
            else
```

```
            {
    /*
     *-------------------------------------------------------------
     * If the original absolute difference value is >= 30, then
     * must pack two more half bytes in additional to the
     * continuation half byte.
     *-------------------------------------------------------------
     */
            if (*byte_partP == 1)
                {
                *byte_maskUCP     = (TBITS) (value - 14);
                *num_comp_bytesP += 1;
                **comp_arrayUCPP  = *byte_maskUCP;
                ++(*comp_arrayUCPP);
                }
             else
                {
                get_4_bits(&tempUC, 1, (TBITS) (value - 14));
                set_4_bits(tempUC, 2, byte_maskUCP);
                *num_comp_bytesP += 1;
                **comp_arrayUCPP  = *byte_maskUCP;
                ++(*comp_arrayUCPP);
                get_4_bits(&tempUC, 2, (TBITS) (value - 14));
                set_4_bits((UTINY) tempUC, 1, byte_maskUCP);
                }
            }
        return(YES);              /* value unpacked successfully*/
    /*-------------------------------------------------------------*/
        }                         /* end of pack_halfbytes      */
    /* UNCMPSIG2.C
     *-------------------------------------------------------------
     * Function to uncompress an array of data compressed by
     * function CMPSIG2. NOTE: compression/uncompression is exact
     * (not lossy) and restores the original data exactly.
     *
     *       uncmpsig2 returns:  1  (i.e. YES or  TRUE) for
     *                           successful uncompress
```

```
    *                           0  (i.e. NO  or FALSE) if
    *                           there is an error
    *
    * input arguments:
    *      comp_arrayUC - unsigned byte array containing the
    *                     compressed representation of the data
    *                     (as created by CMPSIG2.C)
    *
    * output arguments:
    *      numoutP - ptr to number of elements (length) of the
    *                     uncompressed data
    *      array   - data array created by unpacking and
    *                     reconstructing values taken from the
    *                     compressed array comp_arrayUC[].
    *-----------------------------------------------------------
    */
   #include <stdio.h>
   #include <cic.h>

   BOOL uncmpsig2(numoutP, array, comp_arrayUC)
       COUNT *numoutP;          /* number of points in output */
                                /* array                      */
       COUNT array[];           /* array for output signat    */
                                /* data that has reconst from */
                                /* comp_arrayUC by reversing  */
                                /* the compression proc.      */
       TBITS comp_arrayUC[];    /* compressed input array of  */
                                /* sig data                   */
       {
       COUNT i;                 /* loop index                 */
       COUNT kount_bytes;       /* kounts bytes used in       */
                                /* comp_array                 */
       TBITS *comp_arrayUCP;    /* ptr to comp_arrayUC of sig */
                                /* data                       */
       COUNT num_sign_change;   /* counts number of sign      */
                                /* changes                    */
       COUNT value;             /* for general use temporary  */
                                /* storage                    */
```

```
    COUNT byte_part;          /* 1 if upper 4 bits of byte, */
                              /* and 2 if lower 4 bits of   */
                              /* byte                       */
    TBITS byte_maskUC;        /* packed byte representing   */
                              /* different values           */
    COUNT sign_change[256];   /* array of sign change       */
                              /* indices                    */
    COUNT *sign_changeP;      /* pointer to sign change     */
                              /* index array                */
    COUNT sign_first_dif;     /* saves sign of first        */
                              /* difference                 */
    COUNT current_sign;       /* current value of sign      */
    BOOL  unpack_halfbytes(); /* implements main unpacking  */
                              /* logic                      */
/*
 *-------------------------------------------------------------
 * Initializations for compressed array
 *-------------------------------------------------------------
 */
    kount_bytes   = 0;
    comp_arrayUCP = comp_arrayUC;
/*
 *-------------------------------------------------------------
 * First, get the number of items in the array that was
 * compressed.  This value is stored as two bytes.  The first
 * byte is the upper 8 bits of the 16 bit COUNT value for
 * numout, and the second byte is the lower 8 bits.
 *-------------------------------------------------------------
 */
    *numoutP     = 0;
    *numoutP     = *comp_arrayUCP++;
    *numoutP   <<= 8;
    *numoutP     = *numoutP | (*comp_arrayUCP++);
    kount_bytes += 2;
  /*
 *-------------------------------------------------------------
 * Get the number of sign changes
 *-------------------------------------------------------------
```

```
   */
      num_sign_change = (COUNT) (*comp_arrayUCP++);
      ++kount_bytes;
   /*
    *-------------------------------------------------------------
    * Check for error condition.  If the value read in for the
    * number of sign changes is >= 256, then an error has
    * occurred.
    *-------------------------------------------------------------
    */
      if (num_sign_change < 0 || num_sign_change >= 256)
          {
          fprintf(stderr, "\n\nUNCMPSIG -- number of sign
                changes out ");
          fprintf(stderr, "of range (= %d) is >= 256\n\n",
                num_sign_change);
          return(NO);          /* return on ERROR detection   */
          }
   /*
    *-------------------------------------------------------------
    * Initializations for unpacking.
    *-------------------------------------------------------------
    */
      byte_part = 1;
   /*
    *-------------------------------------------------------------
    * Unpack the sign of the first value of the original,
    * uncompressed data.
    *-------------------------------------------------------------
    */
      unpack_halfbytes(&value, &byte_part, &byte_maskUC,
                  &comp_arrayUCP, &kount_bytes);
      if (value == 0)
          current_sign = -1;
      else if (value == 1)
          current_sign =  1;
      else
          {
```

```
    /*
     *-------------------------------------------------------------
     * Error condition since value != 0 (for minus sign) and also
     * != 1 (for plus sign).
     *-------------------------------------------------------------
     */
            fprintf(stderr, "\n\nUNCMPSIG -- initial sign value ");
            fprintf(stderr, "is inconsistent (= %d)\n\n", value);
            return(NO);           /* return on ERROR detection  */
            }
    /*
     *-------------------------------------------------------------
     * Unpack the sign of the first difference value of the
     * original, uncompressed data.
     *-------------------------------------------------------------
     */
        unpack_halfbytes(&value, &byte_part, &byte_maskUC,
                    &comp_arrayUCP,
                &kount_bytes);
        if (value == 0)
            sign_first_dif = -1;
        else if (value == 1)
            sign_first_dif =  1;
        else
            {
    /*
     *-------------------------------------------------------------
     * Error condition since value != 0 (for minus sign) and also
     * != 1 (for plus sign).
     *-------------------------------------------------------------
     */
            fprintf(stderr, "\n\nUNCMPSIG -- initial sign value ");
            fprintf(stderr, "is inconsistent (= %d)\n\n", value);
            return(NO);           /* return on ERROR detection  */
            }
    /*
     *-------------------------------------------------------------
     * Unpack sign change positions (where successive difference
```

```
 * values have different sign).
 *-------------------------------------------------------------
 */
    for (i = 0; i < num_sign_change; ++i)
        {
        unpack_halfbytes(&value, &byte_part, &byte_maskUC,
                    &comp_arrayUCP, &kount_bytes);
        if (i == 0)
            sign_change[0] = value;
        else
            sign_change[i] = sign_change[i - 1] + value;
        }                       /* end of loop to unpack sign */
                                /*change position values      */
/*
 *-------------------------------------------------------------
 * Unpack differences from bytes and read from disk.
 *-------------------------------------------------------------
 */
    sign_changeP = sign_change;/* pointer to sign change    */
                               /*index array                */
    for (i = 0; i < *numoutP; ++i)
        {
        unpack_halfbytes(&value, &byte_part, &byte_maskUC,
                    &comp_arrayUCP, &kount_bytes);
        if (i > 1 && num_sign_change > 0 && i == *sign_changeP)
            {
            current_sign = -current_sign;
            --num_sign_change;
            ++sign_changeP;
            }
        if (i == 0)
            array[i] =  current_sign * value;
        else if (i == 1)
            {
            sign_changeP = sign_change;
            array[i]   = array[i - 1] + sign_first_dif * value;
            current_sign = sign_first_dif;
            }
```

```
        else
            array[i] = array[i - 1] + current_sign * value;
        }                           /* end of loop to unpack and  */
                                    /* reconstruct data           */
/*
 *-------------------------------------------------------------
 * Check for inconsistency in specified size (in bytes) of the
 * input compressed array (comp_array) versus the actual number
 * of bytes used in the uncompress process to create the output
 * uncompressed array.
 *-------------------------------------------------------------
    if (kount_bytes != num_comp_bytes)
        {
        fprintf(stderr, "\n\nUNCMPSIG -- kount_bytes (= %d)
                    != ", kount_bytes);
        fprintf(stderr, "num_comp_bytes (= %d)\n\n",
                    num_comp_bytes);
        return(NO);
        }
 */
/*
 *-------------------------------------------------------------
 * If this point is reached the uncompression process has
 * proceeded normally and no errors have been detected, so
 * return with a "successful completion" status flag.
 *-------------------------------------------------------------
 */
    return(YES);
/*-------------------------------------------------------------
    }                               /* end of function UNCMPSIG2 */
/* UNPACK_HALFBYTES.C
 *-------------------------------------------------------------
 * Function to unpack and reconstruct a COUNT value from half
 * bytes.
 *
 *      pack_halfbytes returns:  1  (i.e. YES or  TRUE) for
 *                                  successful unpacking
 *                               0  (i.e. NO  or FALSE) if
```

```
 *                                    there is an error
 *-------------------------------------------------------------
 */
BOOL unpack_halfbytes(valueP, byte_partP, byte_maskUCP,
          comp_arrayUCPP, kount_bytesP)
    COUNT *valueP;              /* contains uncompressed      */
                                /* value to be returned to    */
                                /* main program               */
    COUNT *byte_partP;          /* 1  if upper 4 bits of byte,*/
                                /* and 2  if lower 4 bits of  */
                                /* byte                       */
    TBITS *byte_maskUCP;        /* packed byte representing   */
                                /* different values           */
    TBITS **comp_arrayUCPP;     /* ptr to comp_arrayUC of sig.*/
                                /* data                       */
    COUNT *kount_bytesP;        /* kounts bytes used in       */
                                /* comp_array                 */
    {
    UTINY tempUC;               /* temporary value            */
    UTINY valueUC;              /* temporary value            */
    BOOL  get_4_bits();         /* function to unpack values  */
                                /* from the upper or lower 4  */
                                /* bits of a byte             */
    BOOL  set_4_bits();         /* function to pack values    */
                                /* into the upper or lower 4  */
                                /* bits of a byte             */
/*
 *-------------------------------------------------------------
 * Unpack 4 bits.  When the value is < 14, it corresponds to a
 * legitimate difference value, and hence the next array value
 * can be calculated by adding the difference (with the correct
 * sign) to the preceding array value.  If the value == 14, the
 * absolute difference value is >= 14 and <= 29, hence another
 * half byte must be read in to compose the absolute
 * difference.  If the value == 15, the absolute difference
 * value is >= 30 and < 269, hence another full byte (i.e., 2
 * half bytes) must be read in to compose the absolute
 * difference.
```

```
*-----------------------------------------------------------
*/
    if (*byte_partP == 1)
        {
        ++(*kount_bytesP);
        *byte_maskUCP = **comp_arrayUCPP;
        ++(*comp_arrayUCPP);
        }
    get_4_bits(&valueUC, *byte_partP, *byte_maskUCP);
    *byte_partP = 3 - *byte_partP;
    if (valueUC < 14)
        *valueP = (COUNT) valueUC;
    else if (valueUC == 14)
        {
        if (*byte_partP == 1)
            {
            *byte_maskUCP = **comp_arrayUCPP;
            ++(*comp_arrayUCPP);
            ++(*kount_bytesP);
            }
        get_4_bits(&valueUC, *byte_partP, *byte_maskUCP);
        *valueP   = (COUNT) (valueUC + 14);
        *byte_partP = 3 - *byte_partP;
        }
    else if (valueUC == 15)
        {
        if (*byte_partP == 1)
            {
            *byte_maskUCP = **comp_arrayUCPP;
            ++(*comp_arrayUCPP);
            ++(*kount_bytesP);
            }

        get_4_bits(&tempUC, *byte_partP, *byte_maskUCP);
        set_4_bits(tempUC, 1, &valueUC);
        *byte_partP = 3 - *byte_partP;
        if (*byte_partP == 1)
            {
```

```
            *byte_maskUCP = **comp_arrayUCPP;
            ++(*comp_arrayUCPP);
            ++(*kount_bytesP);
            }
        get_4_bits(&tempUC, *byte_partP, *byte_maskUCP);
        set_4_bits(tempUC, 2, &valueUC);
        *valueP =  (COUNT) (valueUC + 14);
        *byte_partP     = 3 - *byte_partP;
        }
    else
        {
        *byte_maskUCP = **comp_arrayUCPP;
        ++(*comp_arrayUCPP); ++(*kount_bytesP);
        }
    return(YES);          /* value unpacked successfully     */
/*-------------------------------------------------------------
 */
    }                     /* end of function UNPACK_HALFBYTES */

/* GET_4_BITS.C
 *-------------------------------------------------------------
 * Function that returns the UTINY value corresponding to
 * either the upper 4 bits in the input byte, or the lower 4
 * bits.  Of course this will be a value in the range 0 to 15.
 * The input arguement "position" determines whether the upper
 * or lower 4 bits is taken:
 *
 *      position  =  1    --   upper four bits (or upper
 *                             "nybble")
 *      position  =  2    --   lower four bits (or upper
 *                             "nybble")
 *-------------------------------------------------------------
 */
#include <stdio.h>
#include <cic.h>
BOOL get_4_bits(valueUCP, position, byte_maskUC)
    UTINY *valueUCP;          /* set the four bits to this  */
                              /* value                      */
```

```
    COUNT position;          /* 1  specifies the upper 4    */
                             /* bits, and                   */
    TBITS byte_maskUC;       /* feature mask for template   */
                             /* selection 2 specifies the   */
                             /* lower 4 bits                */
    {
/*
 *-------------------------------------------------------------
 * Return original mask if error condition found
 *-------------------------------------------------------------
 */
    if (position != 1  && position != 2)
        return(NO);
/*
 *-------------------------------------------------------------
 * Get 4 bits (or nybble) specified by the input argument
 * "position"
 *-------------------------------------------------------------
 */
    if (position == 1)
        byte_maskUC >>= 4;
    *valueUCP = (UTINY) (byte_maskUC & 0xf);
                             /* zero the upper four bits    */
    return(YES);
/*-------------------------------------------------------------
 */
    }                        /* end of function GET_4_BITS.C*/
 */

 * SET_4_BITS.C
 *-------------------------------------------------------------
 * Function that sets either the upper 4 bits in the input
 * byte, or the lower 4 bits to a value in the range 0 to 15
 * specified by the input argument "value".
 *
 * The input arguement "position" determines whether the upper
 * or lower 4 bits is set:
 *
```

```
 *     "position  =  1     --   upper four bits (or upper
 *                              "nybble")
 *      position  =  2     --   lower four bits (or upper
 *                              "nybble")
 *
 * NOTE: function returns a boolean YES for normal operation
 * and NO if an error is detected.
 *-------------------------------------------------------------
 */
#include <stdio.h>
#include <cic.h>
BOOL set_4_bits(valueUC, position, byte_maskUCP)
    UTINY valueUC;            /* set the four bits to this  */
                              /* value                      */
    COUNT position;           /* specifies the position of  */
                              /* the two bits               */
    TBITS *byte_maskUCP;      /* feature mask for template  */
                              /* selection                  */
    {
/*
 *-------------------------------------------------------------
 * Return original mask if error condition found
 *-------------------------------------------------------------
 */
    if (valueUC > (UTINY) 15)
        return(NO);
    if (position != 1  && position != 2)
        return(NO);
   /*
 *-------------------------------------------------------------
 * Set bits as specified by the input arguments
 *-------------------------------------------------------------
 */
    if (position == 1)
        {
        *byte_maskUCP  &= ~(0xf0);
                        /* zero upper 4 bits inbyte_maskUCP*/
        *byte_maskUCP  |= (valueUC << 4);
```

```
                                /*  set upper 4 bits to new value */
            }
        else
            {
            *byte_maskUCP  &= ~(0xf);
                                /* zero lower 4 bits in byte_maskUCP*/
            *byte_maskUCP  |= valueUC;
                                /* set upper 4 bits to new value   */
            }
    return(YES);
/*-----------------------------------------------------------------
    }                           /* end of function SET_4_BITS       */
/*
 *-----------------------------------------------------------------
 * This file contains standard type definitions, defined
 * constants, macros, and data structures.
 *-----------------------------------------------------------------
 */
#define FAST    register
#define GLOBAL
#define IMPORT  extern
#define THISFL  extern
#define INTERN  static
#define LOCAL   static

typedef void    VOID;
typedef char    TBOOL, TINY, TEXT;
typedef short   COUNT, SHORT;
typedef int     INT, ARGINT, METACH;
typedef long    LONG;

typedef unsigned char   UTINY, TBITS, UTEXT;
typedef unsigned short  BITS, UCOUNT, USHORT, SIZETYPE;
typedef unsigned int    BYTES, UINT;
typedef unsigned long   ULONG, LBITS;

typedef float   FLOAT;
typedef double  DOUBLE;
```

```
typedef USHORT  ERCTYPE;

typedef enum
    {
    NO  = 0,
    YES = 1
    } BOOL;

typedef enum
    {
    SUCCEED     = 0,
    FAIL        = 1
    } SUCCESS;

#ifndef abs
#define abs(x)    (((x) < 0) ? -(x) : (x))
#endif
#ifndef max
#define max(x,y)  (((x) < (y)) ? (y) : (x))
#endif
#ifndef min
#define min(x,y)  (((x) < (y)) ? (x) : (y))
#endif

#define FOREVER while(YES)

#define EOS ('\0')
```

WE CLAIM:

1. A method for electronically compressing handwritten data expressible as a sequence of data points, the method comprising the steps of:

capturing a parametric representation of data handwritten on a digitizer tablet;

determining local extrema and saving representations of said local extrema as first data points;

determining stroke endpoints and saving representations of said stroke endpoints as second data points; and

calculating curvature locally about selected data points between said second data points and saving only selected representations of said selected data points based on curvature.

2. The method according to claim 1 wherein said parametric representation of the digitizer data is a sequence of x and y coordinate values in a Cartesian coordinate system as a function of time.

3. The method according to claim 1 wherein said local extrema determining step comprises selecting minimum x_i and y_i and maximum x_i and y_i according to the following relationships:

$$\text{minimum} = x_i < x_{i+1} \text{ and } x_i < x_{i-1},$$
$$\text{maximum} = x_i > x_{i+1} \text{ and } x_i > x_{i-1},$$
$$\text{minimum} = y_i < y_{i+1} \text{ and } y_i < y_{i-1},$$
$$\text{maximum} = y_i > y_{i+1} \text{ and } y_i > y_{i-1},$$

where

x_i and y_i over the range $i = 0$ to $i =$ nsamps-2 is the discrete representation of a stroke, and

nsamps is the number of points.

4. The method according to claim 1 wherein said stroke endpoint determining step comprises selecting a first

endpoint of a stroke, x_0 and y_0, and selecting a last endpoint of a stroke, $x_{nsamps-1}$ and $y_{nsamps-1}$,
where

x_i and y_i over the range i = 0 to i = nsamps-1 is the discrete representation of a stroke, and

nsamps is the number of points.

5. The method according to claim 1 wherein said curvature determining step comprises calculating an angle between adjacent data points according to the relationship:

$$c_i = a_{i+d} - a_{i-d},$$

where

$c_i = c_i - 360$ degrees if $c_i > 180$ degrees,

$c_i = c_i + 360$ degrees if $c_i < -180$ degrees,

$a_{i+d} = \arctan(x_{i+d} - x_i, y_{i+d} - y_i)$,

$a_{i-d} = \arctan(x_i - x_{i-d}, y_i - y_{i-d})$,

d is an integer no larger than 4,

x_i and y_i over the range i = d to i = nsamps-1-d is the discrete representation of a stroke, and

nsamps is the number of points.

6. The method according to claim 5 wherein said curvature angle calculating step further comprises discarding data points with a curvature angle less than 5 degrees or greater than 25 degrees.

7. The method according to claim 5 wherein said curvature angle calculating step further comprises discarding data points with a curvature angle less than 25 degrees or greater than 50 degrees.

8. The method according to claim 1 further including a preprocessing step of discarding duplicate data points according to the following relationships:

if $x_i = x_{i-1}$ and $y_i = y_{i-1}$,

then discard x_i and y_i;

if $x_i \neq x_{i-1}$ and $y_i \neq y_{i-1}$,

then save x_i and y_i;

where

x_i and y_i over the range i = 1 to i = nsamps-1 is the discrete representation of a stroke, and

nsamps is the number of points.

9. The method according to claim 1 further including a preprocessing step of discarding data points according to the following procedure:

set j = 1,

calculate $d = ((x_0 - x_j)^2 + (y_0 - y_j)^2)^{\frac{1}{2}}$,

if d < 5, then discard point corresponding to index j, set j = j + 1, and recalculate d;

if d > 5, then shift remaining points down so that the point (x_j, y_j) corresponds to (x_1, y_1), and quit the calculation,

where

x_i and y_i over the range i = 0 to i = nsamps-1 is the discrete representation of a stroke, and

nsamps is the number of points.

10. The method according to claim 1 further including a preprocessing step of discarding data points according to the following procedure:

set j = 2,

calculate $d = ((x_{nsamps-1}-x_{nsamps-j})^2 + (y_{nsamps-1}-y_{nsamps-j})^2)^{\frac{1}{2}}$,

if d < 5, then discard point corresponding to index j, set j = j + 1, and recalculate d;

if d > 5, then shift remaining points down so that the point $(x_{nsamps-j}, y_{nsamps-j})$ corresponds to $(x_{nsamps-2}, y_{nsamps-2})$ and quit the calculation.

11. The method according to claim 1 further including a preprocessing step of dividing data points between said first data points into groups of three adjacent points and calculating the median value of said group according to the following relationships:

x_j = median value of (x_{j-1}, x_j, x_{j+1}), and

y_j = median value of (y_{j-1}, y_j, y_{j+1}).

where

x_i and y_i over the range i = 1 to i = nsamps-2 is the discrete representation of a stroke, and

nsamps is the number of points.

12. The method according to claim 1 further including a preprocessing step of dividing data points between said first data points into groups of three adjacent data points and calculating a representative value for said group according to the following relationships:

$x_j = 0.25x_{j-1} + 0.5x_j + 0.25x_{j+1}$, and

$y_j = 0.25y_{j-1} + 0.5y_j + 0.25y_{j+1}$.

where

x_i and y_i over the range i = 1 to i = nsamps-2 is the discrete representation of a stroke, and

nsamps is the number of points.

13. An apparatus for electronically compressing handwritten data expressible as a sequence of data points, the apparatus comprising:

means for capturing a parametric representation of data handwritten on a digitizer tablet;

means for determining local extrema and saving representations of said local extrema as first data points;

means for determining stroke endpoints and saving representations of said stroke endpoints as second data points; and

means for calculating curvature locally about selected data points between said second data points and saving only selected representations of said selected data points based on curvature.

14. The apparatus according to claim 13 wherein said parametric representation of the digitizer data is a sequence of x and y coordinate values in a Cartesian coordinate system as a function of time.

15. The apparatus according to claim 13 wherein said local extrema determining means comprises means for selecting minimum x_i and y_i and means for selecting maximum x_i and y_i according to the following relationships:

minimum = $x_i < x_{i+1}$ and $x_i < x_{i-1}$,
maximum = $x_i > x_{i+1}$ and $x_i > x_{i-1}$,
minimum = $y_i < y_{i+1}$ and $y_i < y_{i-1}$,
maximum = $y_i > y_{i+1}$ and $y_i > y_{i-1}$,

where

x_i and y_i over the range i = 0 to i = nsamps-2 is the discrete representation of a stroke, and
nsamps is the number of points.

16. The apparatus according to claim 13 wherein said stroke endpoint determining means comprises means for selecting a first endpoint of a stroke, x_0 and y_0, and means for selecting a last endpoint of a stroke, $x_{nsamps-1}$ and $y_{nsamps-1}$, where

x_i and y_i over the range i = 0 to i = nsamps-1 is the discrete representation of a stroke, and
nsamps is the number of points.

17. The apparatus according to claim 13 wherein said curvature determining means comprises means for calculating an angle between adjacent data points according to the relationship:

$$C_i = a_{i+d} - a_{i-d},$$

where

$C_i = C_i - 360$ degrees if $C_i > 180$ degrees,
$C_i = C_i + 360$ degrees if $C_i < -180$ degrees,
$a_{i+d} = \arctan(x_{i+d} - x_i, y_{i+d} - y_i)$,
$a_{i-d} = \arctan(x_i - x_{i-d}, y_i - y_{i-d})$,
d is an integer no larger than 4,
x_i and y_i over the range i = d to i = nsamps-1-d is the discrete representation of a stroke, and
nsamps is the number of points.

18. The apparatus according to claim 17 wherein said curvature angle calculating means further comprises means for discarding data points with a curvature angle less than 5 degrees or greater than 25 degrees.

19. The apparatus according to claim 17 wherein said curvature angle calculating means further comprises means for discarding data points with a curvature angle less than 25 degrees or greater than 50 degrees.

20. The apparatus according to claim 13 further including a preprocessing means for discarding duplicate data points according to the following relationships:

if $x_i = x_{i-1}$ and $y_i = y_{i-1}$,
then discard x_i and y_i;
if $x_i \neq x_{i-1}$ and $y_i \neq y_{i-1}$,
then save x_i and y_i;

where

x_i and y_i over the range i = 1 to i = nsamps-1 is the discrete representation of a stroke, and
nsamps is the number of points.

21. The apparatus according to claim 13 further including a preprocessing means for discarding data points according to the following procedure:

set j = 1,
calculate $d = ((x_0 - x_j)^2 + (y_0 - y_j)^2)^{\frac{1}{2}}$,
if d < 5, then discard point corresponding to index j, set j = j + 1, and recalculate d;
if d > 5, then shift remaining points down so that the point (x_j, y_j) corresponds to (x_1, y_1), and quit the calculation,

where

x_i and y_i over the range i = 0 to i = nsamps-1 is the discrete representation of a stroke, and
nsamps is the number of points.

22. The apparatus according to claim 13 further including a preprocessing means for discarding data points according to the following procedure:

set j = 2,

calculate d = $\{(x_{nsamps-1}-x_{nsamps-j})^2+(y_{nsamps-1}-y_{nsamps-j})^2\}^{\frac{1}{2}}$,

if d < 5, then discard point corresponding to index j, set j = j + 1, and recalculate d;

if d > 5, then shift remaining points down so that the point ($x_{nsamps-j}$, $y_{nsamps-j}$) corresponds to ($x_{nsamps-2}$, $y_{nsamps-2}$) and quit the calculation.

23. The apparatus according to claim 13 further including a preprocessing means for dividing data points between said first data points into groups of three adjacent points and means for calculating the median value of said group according to the following relationships:

x_j = median value of $\{x_{j-1}, x_j, x_{j+1}\}$, and

y_j = median value of $\{y_{j-1}, y_j, y_{j+1}\}$.

where

x_i and y_i over the range i = 1 to i = nsamps-2 is the discrete representation of a stroke, and

nsamps is the number of points.

24. The apparatus according to claim 13 further including a preprocessing means for dividing data points between said first data points into groups of three adjacent data points and means for calculating a representative value for said group according to the following relationships:

$x_j = 0.25x_{j-1} + 0.5x_j + 0.25x_{j+1}$, and

$y_j = 0.25y_{j-1} + 0.5y_j + 0.25y_{j+1}$.

where

x_i and y_i over the range i = 1 to i = nsamps-2 is the discrete representation of a stroke, and

nsamps is the number of points.

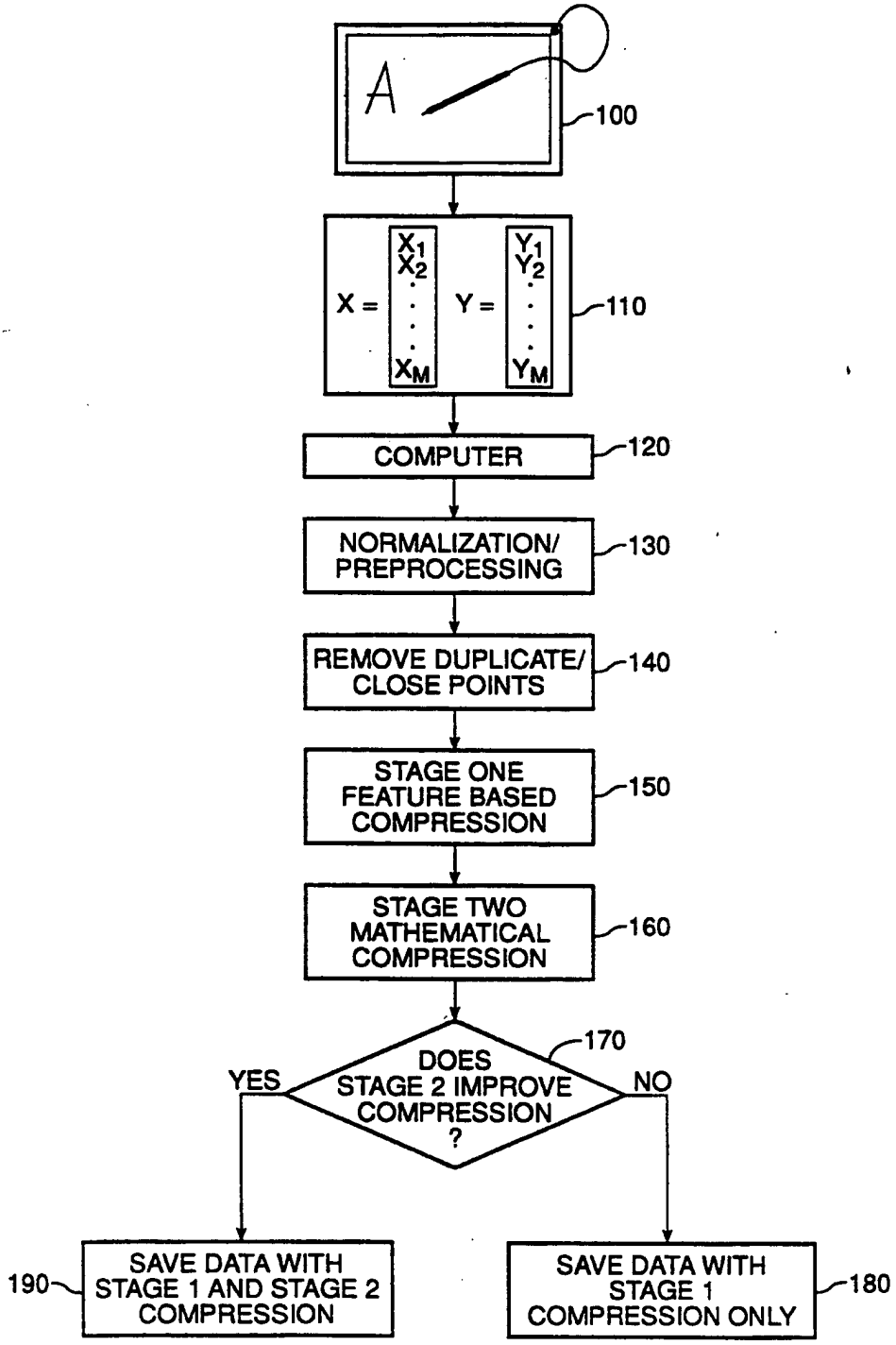

A
100
X = X1 X2 ... XM Y = Y1 Y2 ... YM
110
COMPUTER
120
NORMALIZATION/ PREPROCESSING
130
REMOVE DUPLICATE/ CLOSE POINTS
140
STAGE ONE FEATURE BASED COMPRESSION
150
STAGE TWO MATHEMATICAL COMPRESSION
160
DOES STAGE 2 IMPROVE COMPRESSION ?
170
YES
NO
SAVE DATA WITH STAGE 1 AND STAGE 2 COMPRESSION
190
SAVE DATA WITH STAGE 1 COMPRESSION ONLY
180

FIG. 1

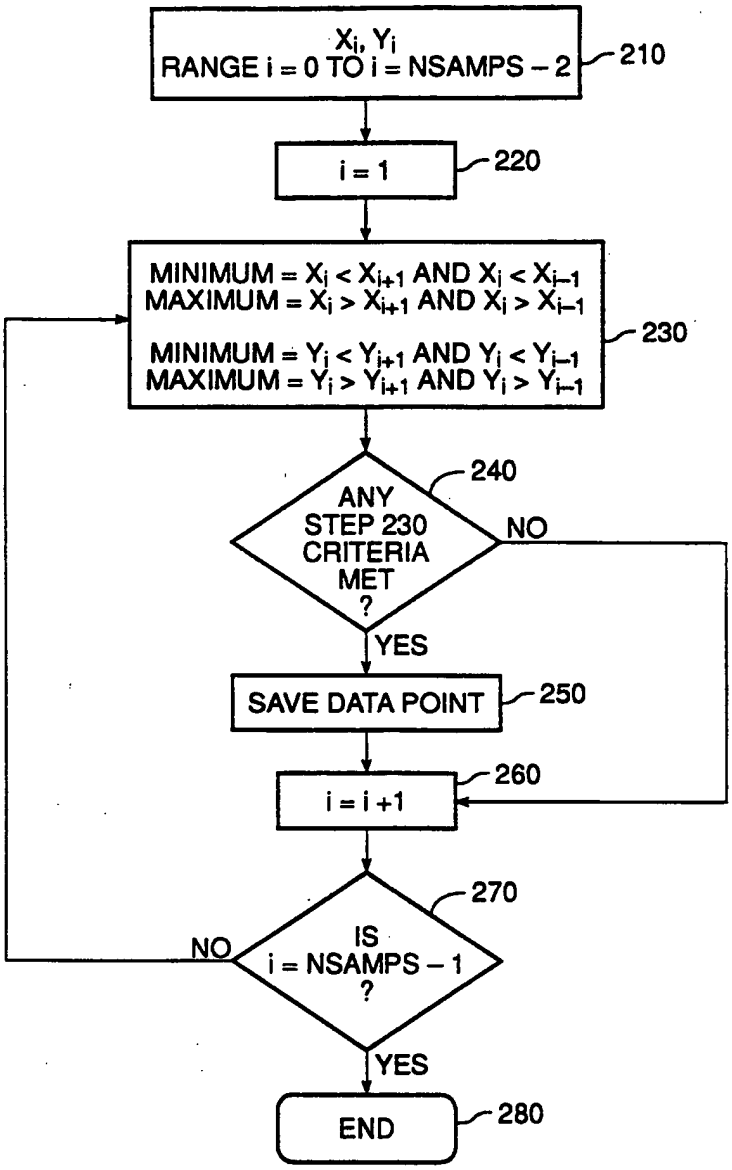

X_i, Y_i
RANGE i = 0 TO i = NSAMPS – 2
210
i = 1
220
MINIMUM = $X_i < X_{i+1}$ AND $X_i < X_{i-1}$
MAXIMUM = $X_i > X_{i+1}$ AND $X_i > X_{i-1}$
MINIMUM = $Y_i < Y_{i+1}$ AND $Y_i < Y_{i-1}$
MAXIMUM = $Y_i > Y_{i+1}$ AND $Y_i > Y_{i-1}$
230
ANY STEP 230 CRITERIA MET ?
240
NO
YES
SAVE DATA POINT
250
i = i +1
260
IS i = NSAMPS – 1 ?
270
NO
YES
END
280

FIG. 2

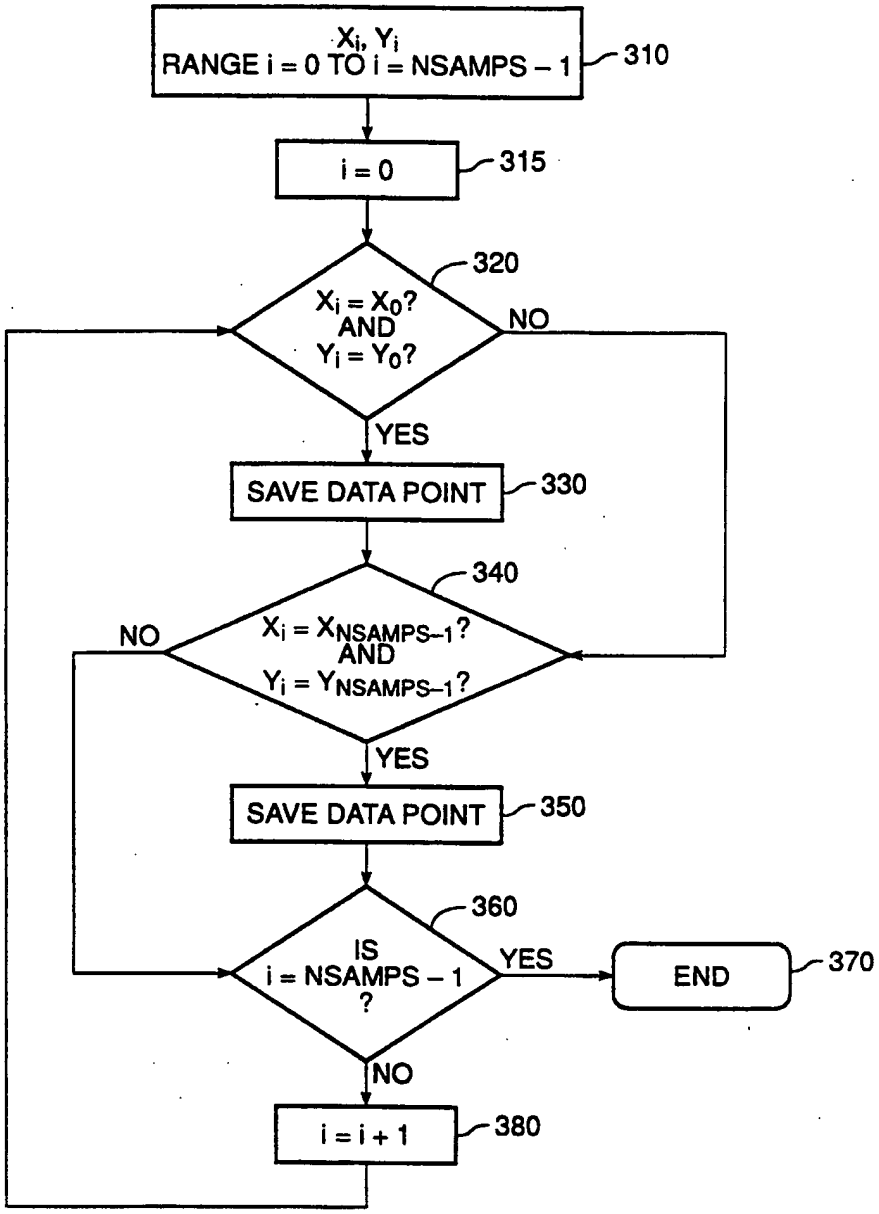

X_i, Y_i
RANGE i = 0 TO i = NSAMPS – 1
310
i = 0
315
320
$X_i = X_0$?
AND
$Y_i = Y_0$?
NO
YES
SAVE DATA POINT
330
340
NO
$X_i = X_{NSAMPS-1}$?
AND
$Y_i = Y_{NSAMPS-1}$?
YES
SAVE DATA POINT
350
360
IS
i = NSAMPS – 1
?
YES
END
370
NO
i = i + 1
380

FIG. 3

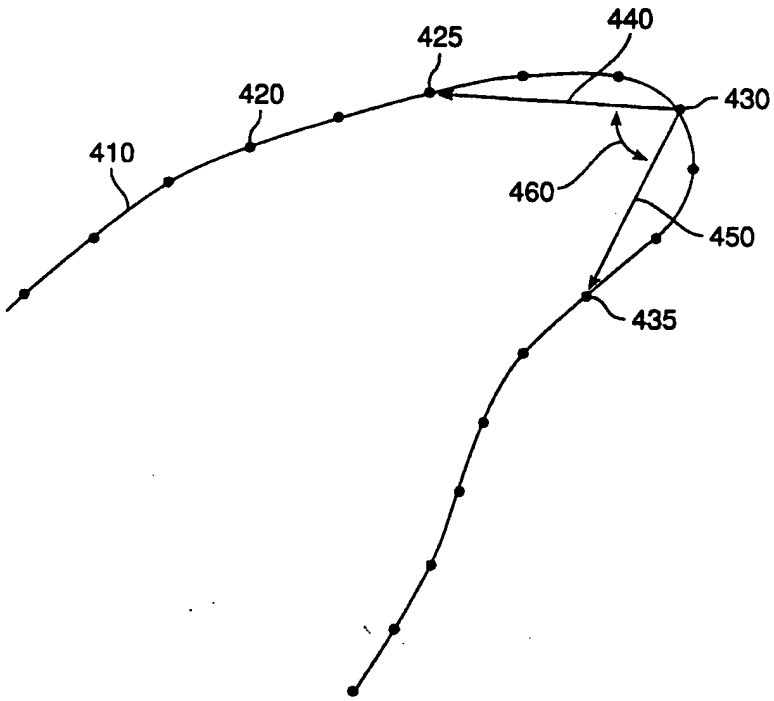


FIG. 4

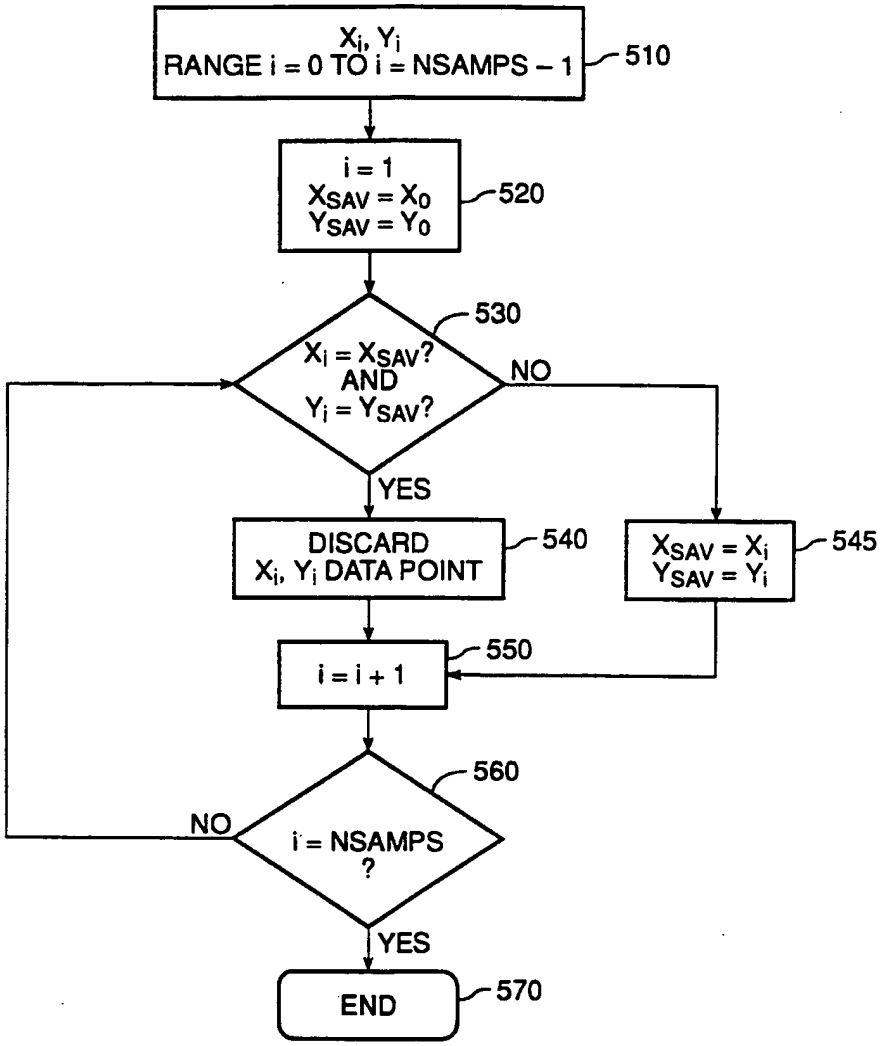

X_i, Y_i
RANGE i = 0 TO i = NSAMPS – 1
510
i = 1
$X_{SAV} = X_0$
$Y_{SAV} = Y_0$
520
530
$X_i = X_{SAV}$?
AND
$Y_i = Y_{SAV}$?
NO
YES
DISCARD
X_i, Y_i DATA POINT
540
$X_{SAV} = X_i$
$Y_{SAV} = Y_i$
545
i = i + 1
550
560
NO
i = NSAMPS
?
YES
END
570

FIG. 5

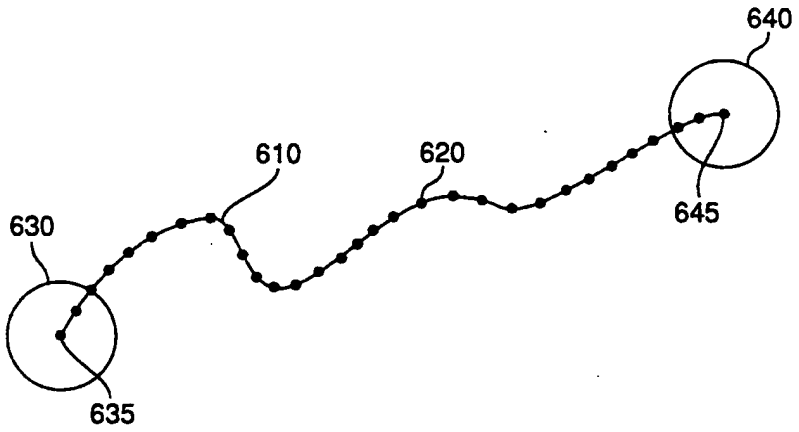


FIG. 6

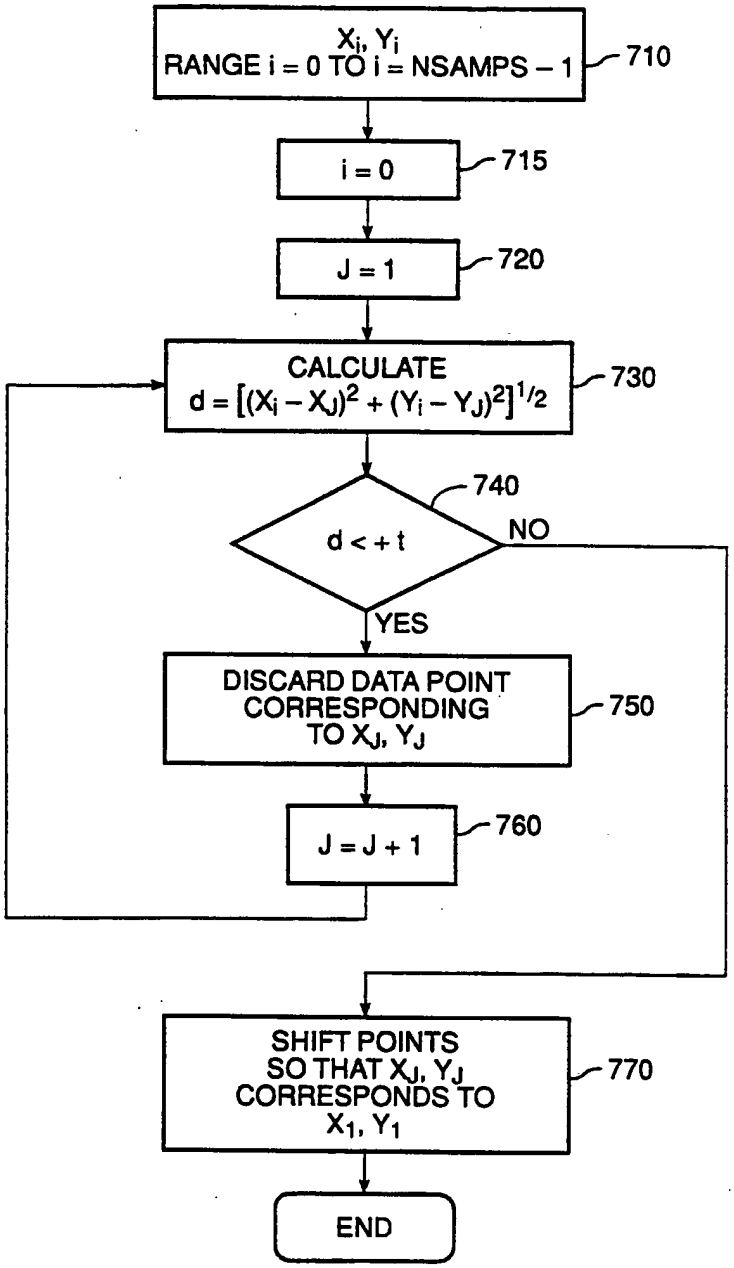

X_i, Y_i
RANGE i = 0 TO i = NSAMPS – 1
710
i = 0
715
J = 1
720
CALCULATE
$d = [(X_i - X_J)^2 + (Y_i - Y_J)^2]^{1/2}$
730
740
d < + t
NO
YES
DISCARD DATA POINT CORRESPONDING TO X_J, Y_J
750
J = J + 1
760
SHIFT POINTS SO THAT X_J, Y_J CORRESPONDS TO X_1, Y_1
770
END

FIG. 7

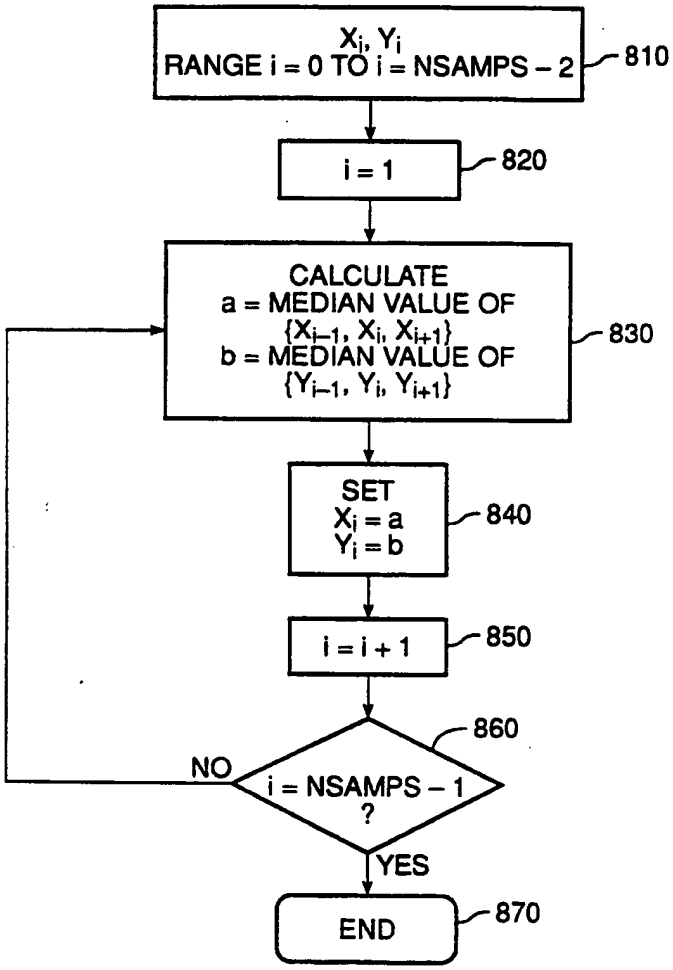

X_i, Y_i
RANGE i = 0 TO i = NSAMPS – 2
810
i = 1
820
CALCULATE
a = MEDIAN VALUE OF
$\{X_{i-1}, X_i, X_{i+1}\}$
b = MEDIAN VALUE OF
$\{Y_{i-1}, Y_i, Y_{i+1}\}$
830
SET
$X_i = a$
$Y_i = b$
840
i = i + 1
850
860
NO
i = NSAMPS – 1
?
YES
END
870

FIG. 8

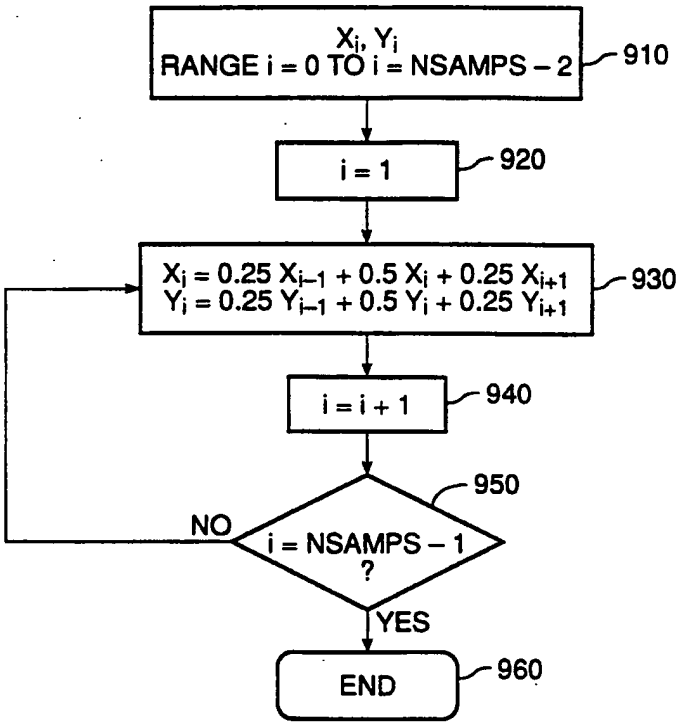

X_i, Y_i
RANGE i = 0 TO i = NSAMPS – 2
910
i = 1
920
$X_i = 0.25\ X_{i-1} + 0.5\ X_i + 0.25\ X_{i+1}$
$Y_i = 0.25\ Y_{i-1} + 0.5\ Y_i + 0.25\ Y_{i+1}$
930
i = i + 1
940
950
NO
i = NSAMPS – 1
?
YES
END
960

FIG. 9

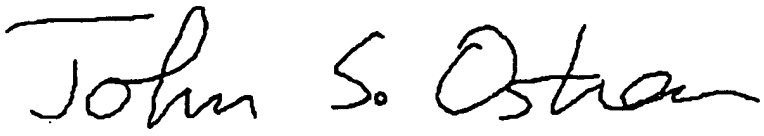


FIG. 10a

FIG. 10b

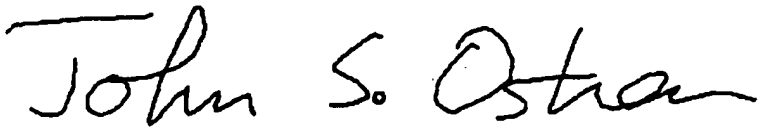


FIG. 10c

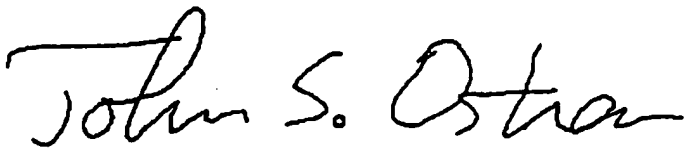


FIG. 11a

FIG. 11b

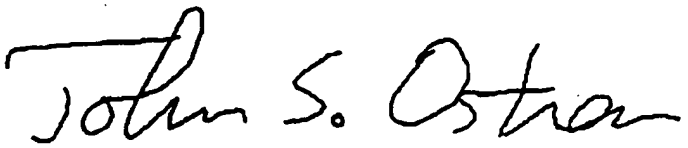


FIG. 11c

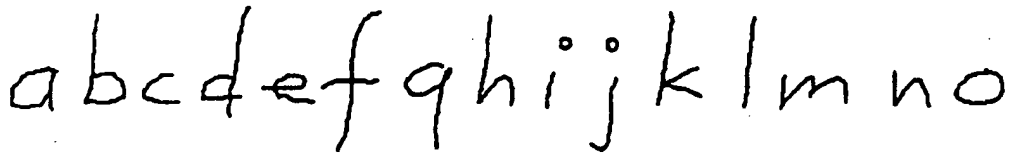


FIG. 12a

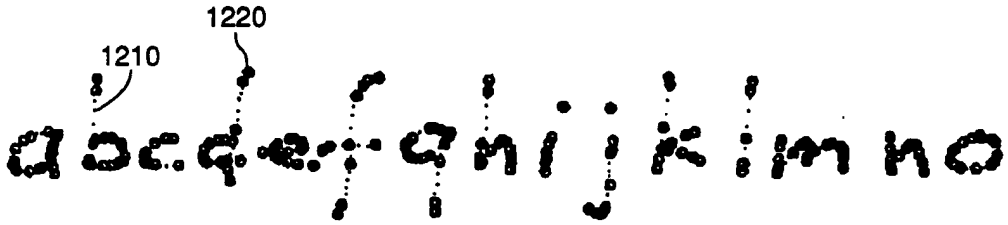


FIG. 12b

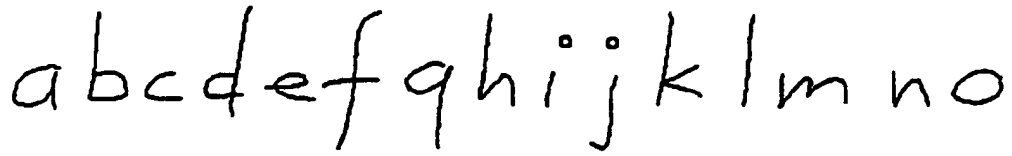


FIG. 12c

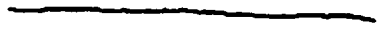


FIG. 13a

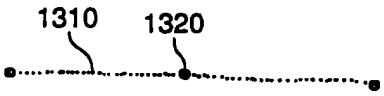


FIG. 13b

FIG. 13c

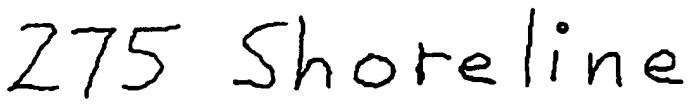


FIG. 14a

FIG. 14b

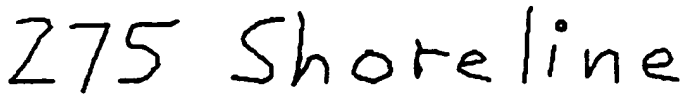


FIG. 14c

FIG. 14d

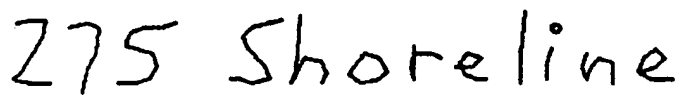


FIG. 14e

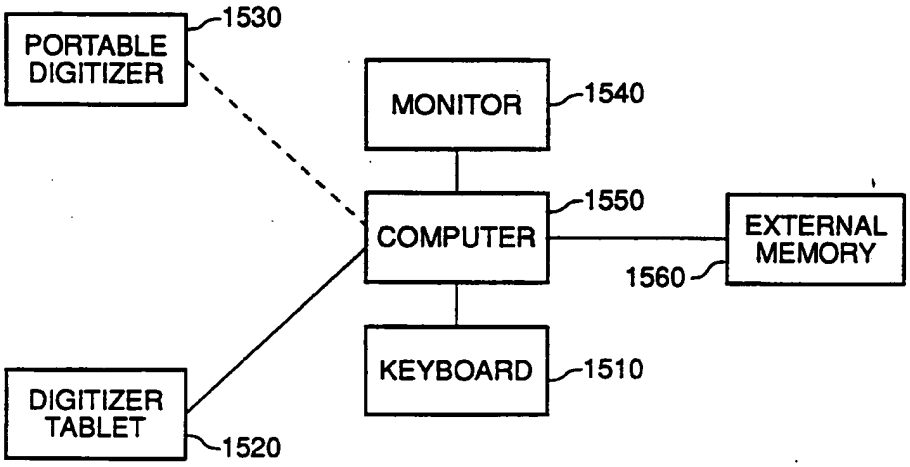

PORTABLE DIGITIZER
1530
MONITOR
1540
COMPUTER
1550
EXTERNAL MEMORY
1560
KEYBOARD
1510
DIGITIZER TABLET
1520

FIG. 15

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US93/06883

A. CLASSIFICATION OF SUBJECT MATTER

IPC(5) :G06F 3/14, 9/00; G06K 9/36, 9/46, 9/00
US CL :382/56, 24; 364/468, 474.03
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 382/56, 24; 364/468, 474.03

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

APS AND JPOABS compress?, handwritten, curvature, endprints data points, angle

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US,A, 4,807,143 (MATSUVRA) 02 February 1989 see fig. 1, 12, col. 3, lines 2-19, and col. 14, lines 32-50.	1-2, 13-14
A	US,A, 4,524,456 (ARAKI et al) 06 June 1985	
A	US,A, 5,023,918 (LIPSCOMB) 06 June 1991	
A	US,A, 5,126,948 (MITCHELL et al) 06 June 1992	

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

*	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be part of particular relevance	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"E"	earlier document published on or after the international filing date	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"&"	document member of the same patent family
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search 23 SEPTEMBER 1993	Date of mailing of the international search report 20 OCT 1993
Name and mailing address of the ISA/US Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231 Facsimile No. NOT APPLICABLE	Authorized officer for MATHEW C. BELLA Telephone No. (703) 308-6720

Form PCT/ISA/210 (second sheet)(July 1992)*

This Page is Inserted by IFW Indexing and Scanning Operations and is not part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☐ BLACK BORDERS
- ☐ IMAGE CUT OFF AT TOP, BOTTOM OR SIDES
- ☐ FADED TEXT OR DRAWING
- ☐ BLURRED OR ILLEGIBLE TEXT OR DRAWING
- ☐ SKEWED/SLANTED IMAGES
- ☐ COLOR OR BLACK AND WHITE PHOTOGRAPHS
- ☐ GRAY SCALE DOCUMENTS
- ☐ LINES OR MARKS ON ORIGINAL DOCUMENT
- ☐ REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY
- ☐ OTHER: _____

IMAGES ARE BEST AVAILABLE COPY.

As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.